COMP9444: Neural Networks and Deep Learning

Week 9d. Course Review

Alan Blair

School of Computer Science and Engineering

July 30, 2025

Planned Topics

Week 0: Refreshers: Python, Numpy, Matplotlib, Google Colab

Week 1: Neuroanatomy, Perceptrons, Multi-Layer Perceptrons, Backprop

Week 2: Probability, Generalisation & Overfitting, PyTorch

Week 3: Cross Entropy, SoftMax, Weight Decay, Momentum, Hidden Unit Dynamics

Week 4: Convolutional Neural Networks, Image Processing

Week 5: Recurrent Neural Networks (RNN), Long Short-Term Memory Network (LSTM), Gated Recurrent Unit (GRU)

Week 6: Flexibility Week

Week 7: Word Vectors, Language Processing, Large Language Models (LLMs)

Week 8: Reinforcement Learning, TD-learning, Q-learning, Policy Learning, Deep RL

Week 9: Autoencoders, Adversarial Training, Multimodal Learning

Week 10: Generative Artificial Intelligence (GenAI), Review (optional)

Assessments

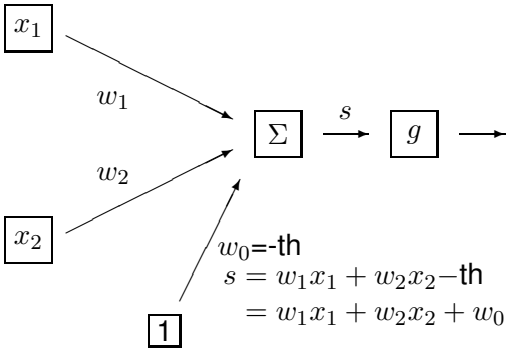
Assessments consist of:

- Assignment (individual): 20%
- Class Participation (individual): 5%
- Group Project (group-based): 30%
- Final Exam (in-person, invigilated): 45%

Due dates:

- Assignment: Due Week 5
- Group Project: Due Week 10
- Final Exam: UNSW Exam Period

McCulloch & Pitts Model of a Single Neuron



x_1, x_2 are inputs

w_1, w_2 are synaptic weights

th is a threshold

w_0 is a **bias** weight

g is transfer function

Perceptron Learning Rule

Adjust the weights as each input is presented.

recall: $s = w_1 x_1 + w_2 x_2 + w_0$

if $g(s) = 0$ but should be 1,

$$\begin{aligned} w_k &\leftarrow w_k + \eta\, x_k \\ w_0 &\leftarrow w_0 + \eta \end{aligned}$$

$$\text{so} \quad s \leftarrow s + \eta\,(1 + \sum_k x_k^2\,)$$

if $g(s) = 1$ but should be 0,

$$\begin{aligned} w_k &\leftarrow w_k - \eta\, x_k \\ w_0 &\leftarrow w_0 - \eta \end{aligned}$$

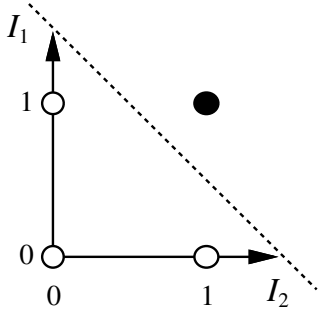
$$\text{so} \quad s \leftarrow s - \eta\,(1 + \sum_k x_k^2\,)$$

otherwise, weights are unchanged. ($\eta > 0$ is called the **learning rate**)

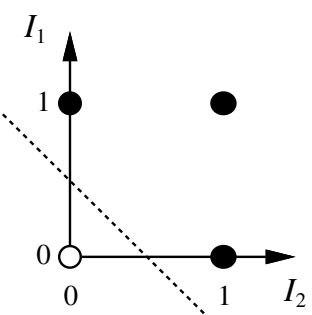
Theorem: This will eventually learn to classify the data correctly, as long as they are **linearly separable**.

Limitations of Perceptrons

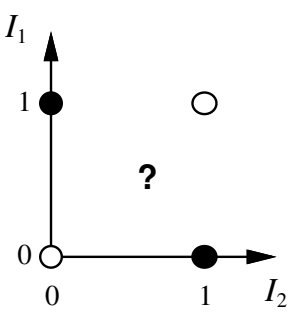
Problem: many useful functions are not linearly separable (e.g. XOR)



(a) I_1 **and** I_2



(b) I_1 **or** I_2



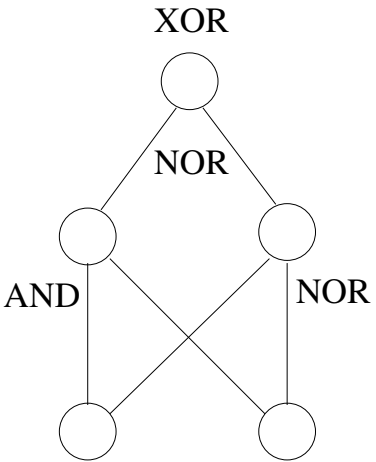
(c) I_1 **xor** I_2

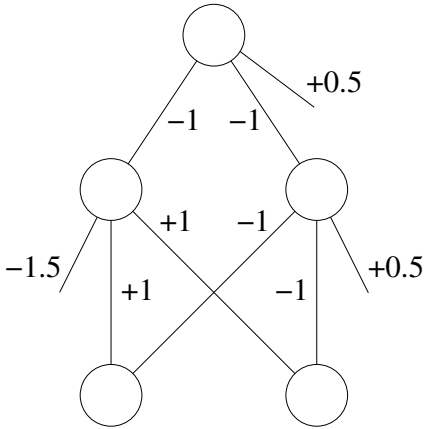
Possible solution:

x_1 XOR x_2 can be written as: (x_1 AND x_2) NOR (x_1 NOR x_2)

Recall that AND, OR and NOR can be implemented by perceptrons.

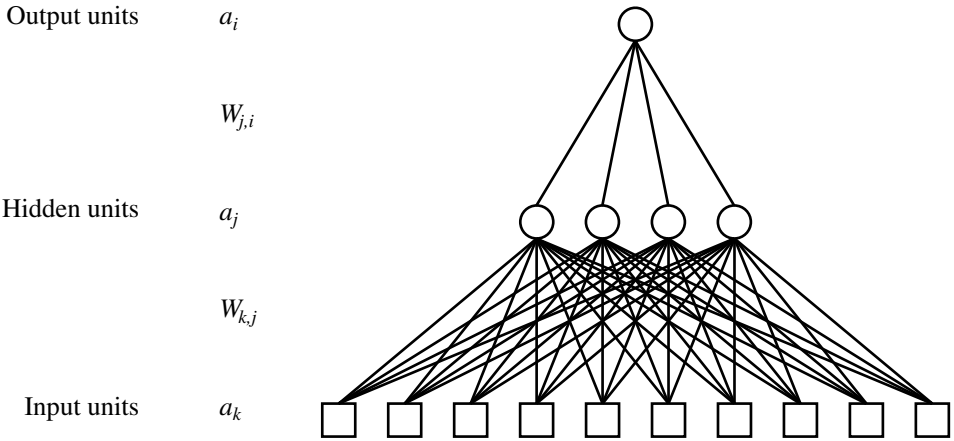
Multi-Layer Neural Networks





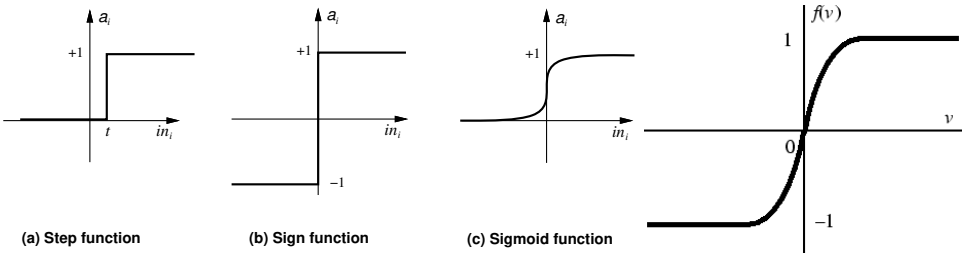
Problem: How can we train it to learn a new function? (credit assignment)

Two-Layer Neural Network



Normally, the numbers of input and output units are fixed,
but we can choose the number of hidden units.

Continuous Activation Functions



(a) Step function (b) Sign function (c) Sigmoid function

Key Idea: Replace the (discontinuous) step function with a differentiable function, such as the sigmoid:

$$g(s) = \frac{1}{1 + e^{-s}}$$

or hyperbolic tangent

$$g(s) = \tanh(s) = \frac{e^{s} - e^{-s}}{e^{s} + e^{-s}} = 2\left(\frac{1}{1 + e^{-2s}}\right) - 1$$

Gradient Descent

Recall that the **loss** function E is (half) the sum over all input patterns of the square of the difference between actual output and target output

$$E = \frac{1}{2} \sum_i (z_i - t_i)^2$$

The aim is to find a set of weights for which E is very low.

If the functions involved are smooth, we can use multi-variable calculus to adjust the weights in such a way as to take us in the steepest downhill direction.

$$w \leftarrow w - \eta \frac{\partial E}{\partial w}$$

Parameter η is called the **learning rate**.

Chain Rule

If, say

$$y = y(u)$$

$$u = u(x)$$

Then

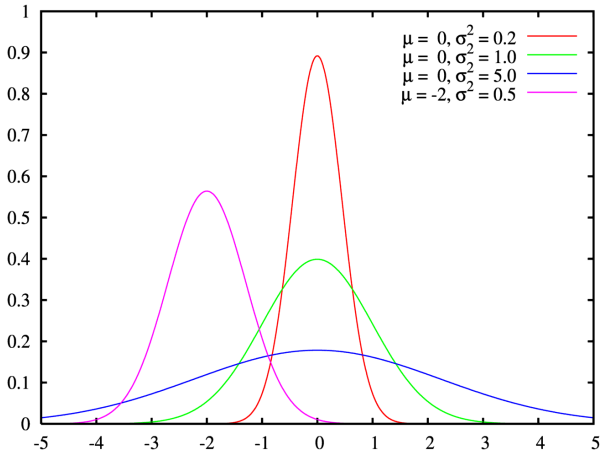
$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\,\frac{\partial u}{\partial x}$$

This principle can be used to compute the partial derivatives in an efficient and localized manner. Note that the transfer function must be differentiable (usually sigmoid, or tanh).

Note: if $z(s) = \dfrac{1}{1+e^{-s}}, \qquad z'(s) = z(1-z).$

if $z(s) = \tanh(s), \qquad z'(s) = 1 - z^2.$

Gaussian Distribution



μ = mean

σ = standard deviation

$$P_{\mu,\sigma}(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2\sigma^2}$$

Bayes' Rule

The formula for conditional probability can be manipulated to find a relationship when the two variables are swapped:

$$P(A \wedge B) = P(A \mid B)P(B) = P(B \mid A)P(A)$$

$$\rightarrow \textbf{Bayes' rule} \quad P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

This is often useful for assessing the probability of an underlying **Cause** after an **Effect** has been observed:

$$P(\text{Cause}|\text{Effect}) = \frac{P(\text{Effect}|\text{Cause})P(\text{Cause})}{P(\text{Effect})}$$

Entropy and KL-Divergence

The **entropy** of a discrete probability distribution $p = \langle p_1, \ldots, p_n \rangle$ is

$$H(p) = \sum_{i=1}^{n} p_i \left(-\log_2 p_i\right)$$

Given two probability distributions $p = \langle p_1, \ldots, p_n \rangle$ and $q = \langle q_1, \ldots, q_n \rangle$ on the same set Ω, the **Kullback-Leibler Divergence** between p and q is

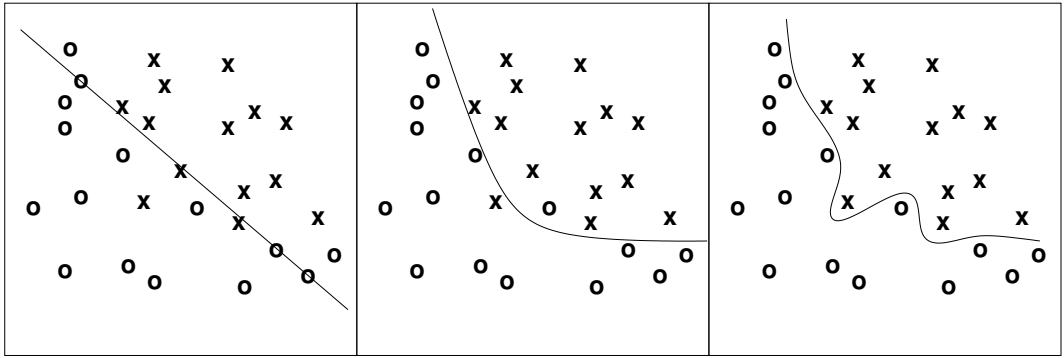
$$D_{\mathrm{KL}}(p \,\|\, q) = \sum_{i=1}^{n} p_i \left(\log_2 p_i - \log_2 q_i\right)$$

KL-Divergence is like a “distance” from one probability distribution to another.
But, it is not symmetric.

$$D_{\mathrm{KL}}(p \,\|\, q) \;\neq\; D_{\mathrm{KL}}(q \,\|\, p)$$

Ockham's Razor

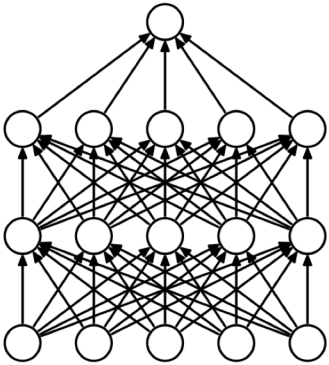
"The most likely hypothesis is the **simplest** one consistent with the data."



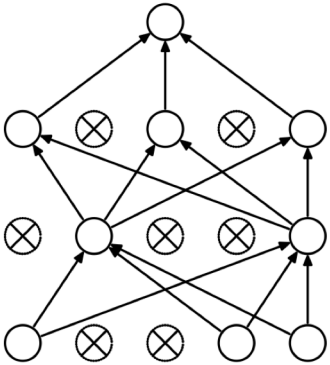
inadequate | good compromise | over-fitting

Since there can be **noise** in the measurements, in practice we need to make a tradeoff between simplicity of the hypothesis and how well it fits the data.

Dropout



(a) Standard Neural Net



(b) After applying dropout.

For each minibatch, randomly choose a subset of nodes to not be used.
Each node is chosen with some fixed probability (usually, one half).

Cross Entropy

For **function approximation**, we normally use the **Sum Squared Error** (SSE) loss:

$$E = \frac{1}{2}\sum_i (z_i - t_i)^2$$

where z_i is the output of the network, and t_i is the target output.

However, for **classification** tasks, where the target output t_i is either 0 or 1, it is more logical to use the **Cross Entropy** loss:

$$E = \sum_i \big(-t_i \log(z_i) - (1 - t_i)\log(1 - z_i)\big)$$

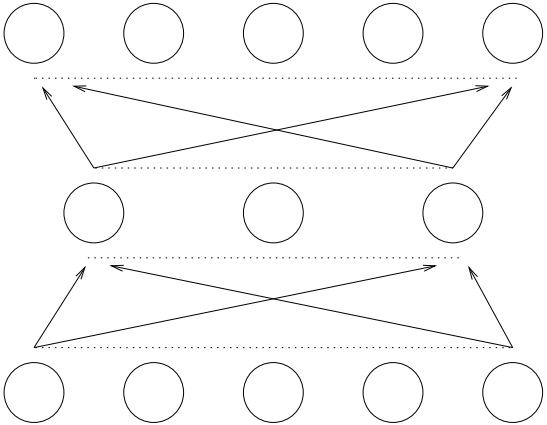
The motivation for these loss functions can be explained using the mathematical concept of **Maximum Likelihood**.

Softmax

- classification task with N classes
- neural network with N outputs $z_1, \ldots, z_N$
- assume the network's estimate for the probability of the correct class being j is proportional to $\exp(z_j)$
- because the probabilites must add up to 1, we need to **normalize** by dividing by their sum:

$$\begin{aligned} \text{Prob}(i) &= \frac{\exp(z_i)}{\sum_{j=1}^{N} \exp(z_j)} \\ \log \text{Prob}(i) &= z_i - \log \sum_{j=1}^{N} \exp(z_j) \end{aligned}$$

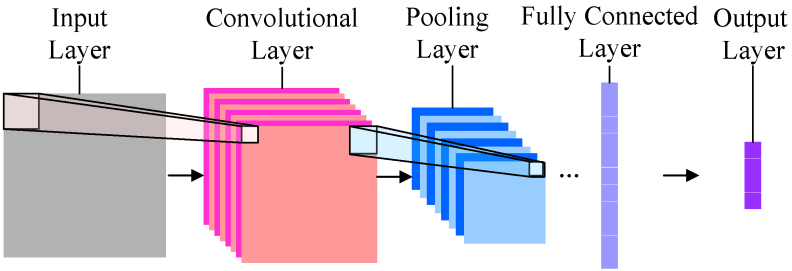
Encoder Networks



Inputs	Outputs
10000	10000
01000	01000
00100	00100
00010	00010
00001	00001

- identity mapping through a bottleneck
- also called N–M–N task
- used to investigate hidden unit representations

Convolutional Network Components



- **convolution layers**: extract shift-invariant features from the previous layer
- **subsampling or pooling layers**: combine the activations of multiple units from the previous layer into one unit
- **fully connected layers**: collect spatially diffuse information
- **output layer**: choose between classes

Weight Initialization

In order to have healthy forward and backward propagation, each term in the product must be approximately equal to 1. Any deviation from this could cause the activations to either vanish or saturate, and the differentials to either decay or explode exponentially.

$$\mathrm{Var}[z] \simeq \Big(\prod_{i=1}^{D} G_0\, n_i^{\mathrm{in}} \mathrm{Var}[w^{(i)}]\Big)\mathrm{Var}[x]$$

$$\mathrm{Var}[\frac{\partial}{\partial x}] \simeq \Big(\prod_{i=1}^{D} G_1\, n_i^{\mathrm{out}} \mathrm{Var}[w^{(i)}]\Big)\mathrm{Var}[\frac{\partial}{\partial z}]$$

We therefore choose the initial weights $\{w_{jk}^{(i)}\}$ in each layer (i) such that

$$G_1 n_i^{\mathrm{out}} \mathrm{Var}[w^{(i)}] = 1$$

Batch Normalization

We can **normalize** the activations $x_k^{(i)}$ of node k in layer (i) relative to the mean and variance of those activations, calculated over a mini-batch of training items:

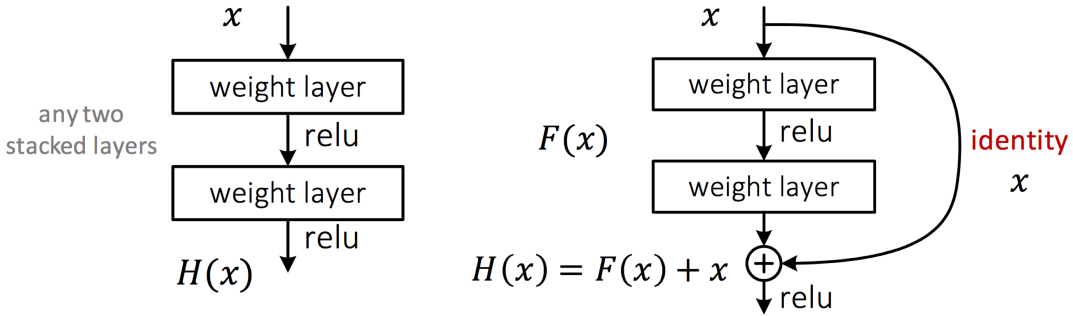
$$\hat{x}_k^{(i)} = \frac{x_k^{(i)} - \mathrm{Mean}[x_k^{(i)}]}{\sqrt{\mathrm{Var}[x_k^{(i)}]}}$$

These activations can then be shifted and re-scaled to

$$y_k^{(i)} = \beta_k^{(i)} + \gamma_k^{(i)} \hat{x}_k^{(i)}$$

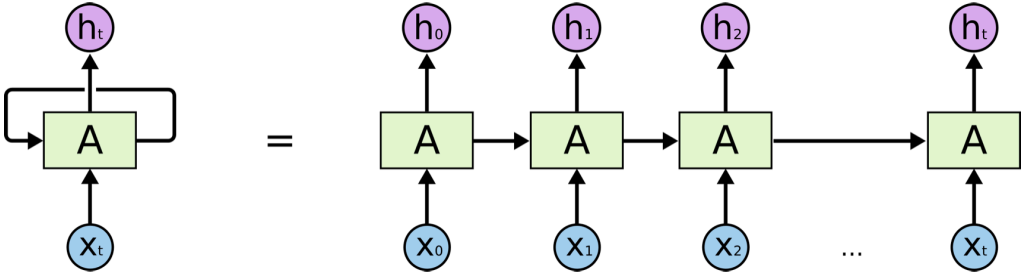
$\beta_k^{(i)}, \gamma_k^{(i)}$ are additional parameters, for each node, which are trained by backpropagation along with the other parameters (weights) in the network.
After training is complete, $\mathrm{Mean}[x_k^{(i)}]$ and $\mathrm{Var}[x_k^{(i)}]$ are either pre-computed on the entire training set, or updated using running averages.

Residual Networks



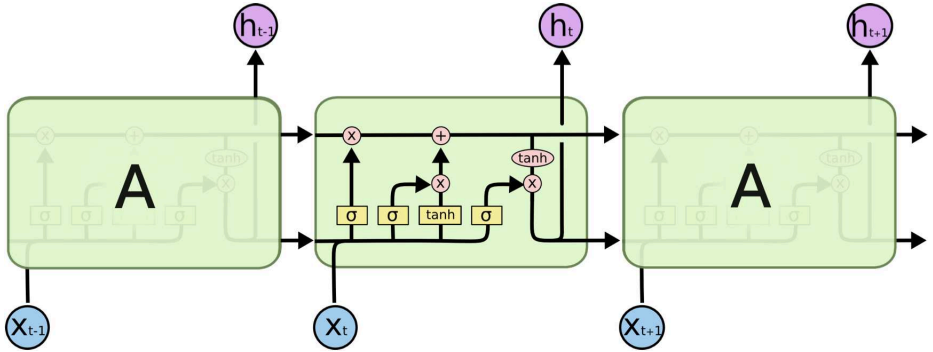
Idea: Take any two consecutive stacked layers in a deep network and add a “skip” connection which bipasses these layers and is added to their output.

Back Propagation Through Time



- we can “unroll” a recurrent architecture into an equivalent feedforward architecture, with shared weights
- applying backpropagation to the unrolled architecture is reffered to as “backpropagation through time”
- we can backpropagate just one timestep, or a fixed number of timesteps, or all the way back to beginning of the sequence

Long Short Term Memory

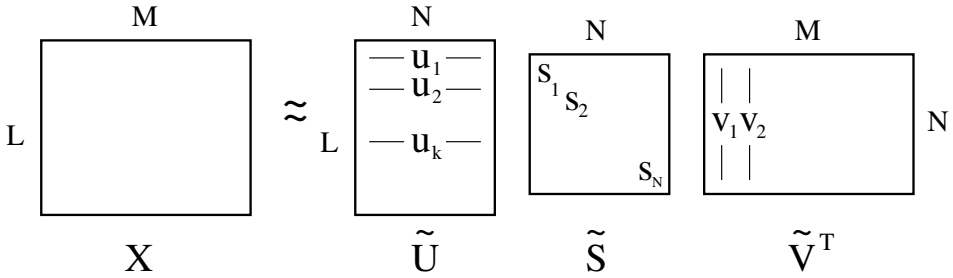


LSTM – context layer is modulated by three gating mechanisms: forget gate, input gate and output gate.

`http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

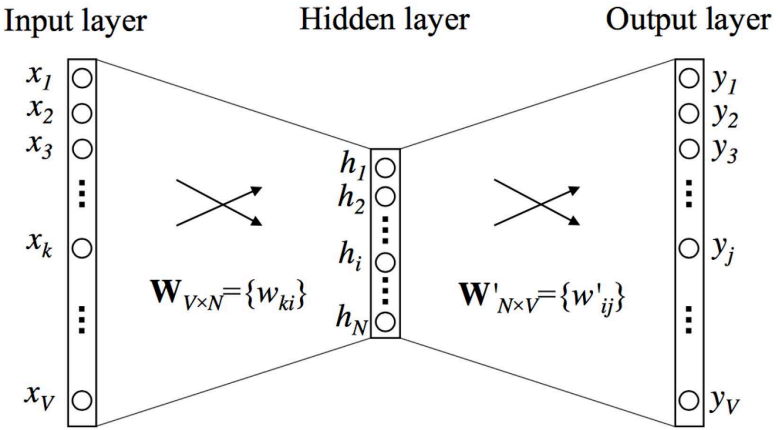
Singular Value Decomposition

Co-occurrence matrix $X_{(L\times M)}$ can be decomposed as $\mathrm{X} = \mathrm{U}\,\mathrm{S}\,\mathrm{V}^{\mathrm{T}}$ where $U_{(L\times L)}$, $V_{(M\times M)}$ are unitary (all columns have unit length) and $S_{(L\times M)}$ is diagonal, with diagonal entries $s_1 \geq s_2 \geq \ldots \geq s_M \geq 0$

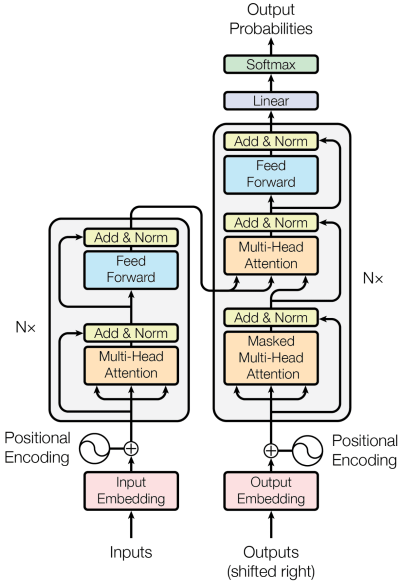


We can obtain an approximation for X of rank $N < M$ by truncating U to $\tilde{\mathrm{U}}_{(L\times N)}$, S to $\tilde{\mathrm{S}}_{(N\times N)}$ and V to $\tilde{\mathrm{V}}_{(N\times M)}$. The kth row of $\tilde{\mathrm{U}}$ then provides an N-dimensional vector representing the k^{th} word in the vocabulary.

word2vec


Input layer
Hidden layer
Output layer
x_1
x_2
x_3
x_k
x_V
h_1
h_2
h_i
h_N
y_1
y_2
y_3
y_j
y_V
$\mathbf{W}_{V\times N}=\{w_{ki}\}$
$\mathbf{W}'_{N\times V}=\{w'_{ij}\}$

Transformer



Reinforcement Learning Framework

- An agent interacts with its environment.
- There is a set $\mathcal{S}$ of **states** and a set $\mathcal{A}$ of **actions**.
- At each time step t, the agent is in some state s_t.
  It must choose an action a_t, whereupon it goes into state $s_{t+1} = \delta(s_t, a_t)$ and receives reward $r_t = \mathcal{R}(s_t, a_t)$
- Agent has a **policy** $\pi : \mathcal{S} \to \mathcal{A}$. We aim to find an **optimal** policy π^* which maximizes the cumulative reward.
- In general, δ, $\mathcal{R}$ and π can be multi-valued, with a random element, in which case we write them as (conditional) probability distributions:

$$\delta(s_{t+1} = s \mid s_t, a_t) \qquad \mathcal{R}(r_t = r \mid s_t, a_t) \qquad \pi(a_t = a \mid s_t)$$

Temporal Difference Learning

Let's first assume that $\mathcal{R}$ and δ are deterministic. Then the (true) value $V^*(s)$ of the current state s should be equal to the immediate reward plus the discounted value of the next state

$$V^*(s) = \mathcal{R}(s, a) + \gamma\, V^*(\delta(s, a))$$

We can turn this into an update rule for the estimated value, i.e.

$$V(s_t) \leftarrow r_t + \gamma\, V(s_{t+1})$$

If $\mathcal{R}$ and δ are stochastic (multi-valued), it is not safe to simply replace $V(s)$ with the expression on the right hand side. Instead, we move its value fractionally in this direction, proportional to a learning rate η

$$V(s_t) \leftarrow V(s_t) + \eta\,[\,r_t + \gamma\, V(s_{t+1}) - V(s_t)\,]$$

Q-Learning

For a deterministic environment, π^*, Q^* and V^* are related by

$$\pi^*(s) = \text{argmax}_a \, Q^*(s,a)$$
$$Q^*(s,a) = \mathcal{R}(s,a) + \gamma \, V^*(\delta(s,a))$$
$$V^*(s) = \max_b \, Q^*(s,b)$$

So

$$Q^*(s,a) = \mathcal{R}(s,a) + \gamma \, \max_b Q^*(\delta(s,a),b)$$

This allows us to iteratively approximate Q by

$$Q(s_t,a_t) \leftarrow r_t + \gamma \, \max_b Q(s_{t+1},b)$$

If the environment is stochastic, we instead write

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \eta \, [\, r_t + \gamma \, \max_b \, Q(s_{t+1},b) - Q(s_t,a_t)]$$

Policy Gradients

Policy Gradients are an alternative to Evolution Strategy, which use gradient ascent rather than random updates.

Let's first consider episodic games. The agent takes a sequence of actions

$$a_1 \; a_2 \; \ldots \; a_t \; \ldots \; a_m$$

At the end it receives a reward r_{total}. We don't know which actions contributed the most, so we just reward all of them equally. If r_{total} is high (low), we change the parameters to make the agent more (less) likely to take the same actions in the same situations. In other words, we want to increase (decrease)

$$\log \prod_{t=1}^{m} \pi_\theta(a_t|s_t) = \sum_{t=1}^{m} \log \pi_\theta(a_t|s_t)$$

Actor Critic Algorithm

for each trial

  sample a_0 *from* $\pi(a|s_0)$

  for each timestep t *do*

    sample reward r_t *from* $\mathcal{R}(r \mid s_t, a_t)$

    sample next state s_{t+1} *from* $\delta(s \mid s_t, a_t)$

    sample action a_{t+1} *from* $\pi(a \mid s_{t+1})$

    $\frac{dE}{dQ} = -[r_t + \gamma\, Q_w(s_{t+1}, a_{t+1}) - Q_w(s_t, a_t)]$

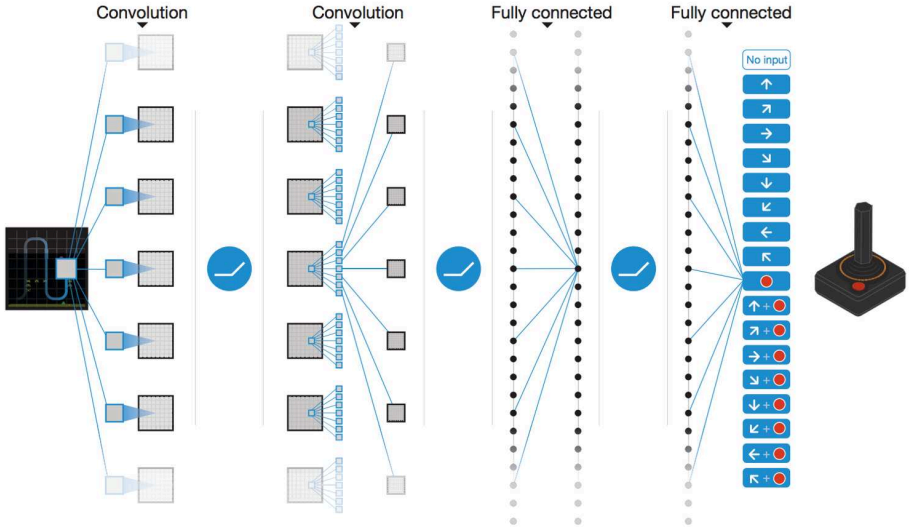
    $\theta \leftarrow \theta + \eta_\theta\, Q_w(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t \mid s_t)$

    $w \leftarrow w - \eta_w \frac{dE}{dQ}\, \nabla_w\, Q_w(s_t, a_t)$

  end

end

Deep Q-Network



Asynchronous Advantage Actor Critic

- use policy network to choose actions
- learn a parameterized Value function $V_u(s)$ by TD-Learning
- estimate Q-value by n-step sample

$$Q(s_t, a_t) = r_{t+1} + \gamma\, r_{t+2} + \ldots + \gamma^{n-1} r_{t+n} + \gamma^n V_u(s_{t+n})$$

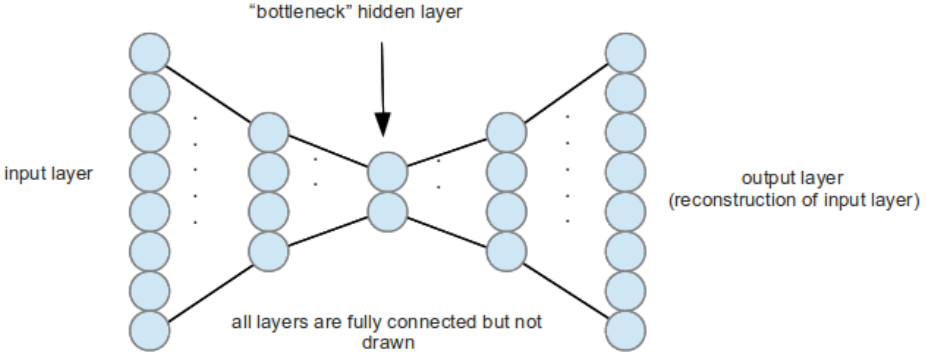
- update policy π_θ by

$$\theta \leftarrow \theta + \eta_\theta \left[Q(s_t, a_t) - V_u(s_t)\right] \nabla_\theta \log \pi_\theta(a_t \mid s_t)$$

- update Value function my minimizing

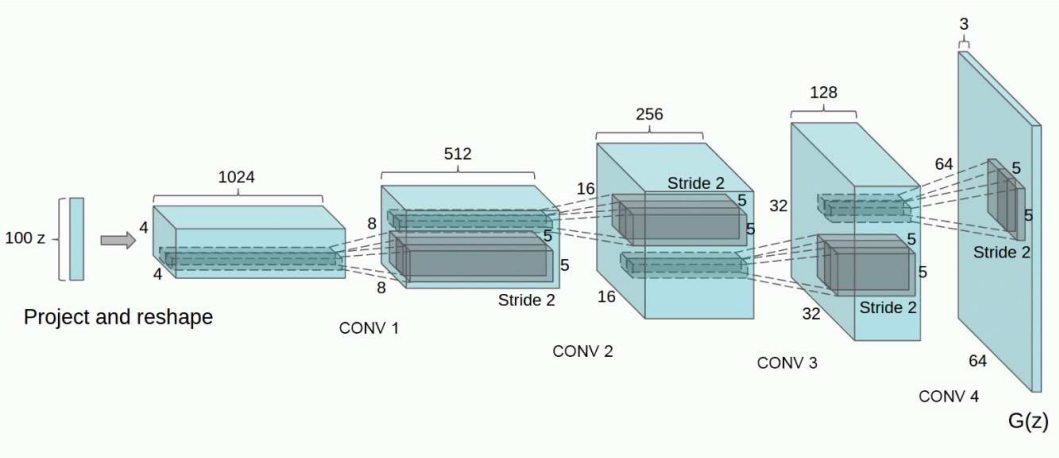
$$\left[Q(s_t, a_t) - V_u(s_t)\right]^2$$

Autoencoder Networks



- output is trained to reproduce the input as closely as possible
- activations normally pass through a bottleneck, so the network is forced to compress the data in some way
- Autoencoders can be used to generate "fake" items, or to automatically extract abstract features from the input

(De-)Convolutional Network to Generate Images



Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks (Radford et al., 2016)

Variational Autoencoder

Instead of producing a single z for each $x^{(i)}$, the encoder (with parameters ϕ) can be made to produce a mean $\mu_{z|x^{(i)}}$ and standard deviation $\sigma_{z|x^{(i)}}$
This defines a conditional (Gaussian) probability distribution $q_\phi(z|x^{(i)})$
We then train the system to maximize

$$\mathbf{E}_{z\sim q_\phi(z|x^{(i)})}[\log\, p_\theta(x^{(i)}|z)\,] \quad - \quad D_{\mathrm{KL}}\big(q_\phi(z|x^{(i)})\|p(z)\big)$$

- the first term enforces that any sample z drawn from the conditional distribution $q_\phi(z|x^{(i)})$ should, when fed to the decoder, produce somthing approximating $x^{(i)}$
- the second term encourages $q_\phi(z|x^{(i)})$ to approximate $p(z)$
- in practice, the distributions $q_\phi(z|x^{(i)})$ for various $x^{(i)}$ will occupy complementary regions within the overall distribution $p(z)$

Generative Adversarial Networks

Generative Adversarial Networks

Generator (Artist) G_θ and Discriminator (Critic) D_ψ are both Deep Convolutional Neural Networks.

Generator $G_\theta : z \mapsto x$, with parameters θ, generates an image x from latent variables z (sampled from a standard Normal distribution).

Discriminator $D_\psi : x \mapsto D_\psi(x) \in (0, 1)$, with parameters ψ, takes an image x and estimates the probability of the image being real.

Generator and Discriminator play a 2-player zero-sum game to compute:

$$\min_\theta \max_\psi \Big(\mathbf{E}_{x \sim p_{\text{data}}} \big[\log D_\psi(x) \big] + \mathbf{E}_{z \sim p_{\text{model}}} \big[\log \big(1 - D_\psi(G_\theta(z)) \big) \big] \Big)$$

Discriminator tries to maximize the bracketed expression,
Generator tries to minimize it.

Generative Adversarial Networks

Alternate between:

Gradient ascent on Discriminator:

$$\max_{\psi}\Big(\mathbf{E}_{x\sim p_{\text{data}}}\big[\log D_{\psi}(x)\big]+\mathbf{E}_{z\sim p_{\text{model}}}\big[\log\big(1-D_{\psi}(G_{\theta}(z))\big)\big]\Big)$$

Gradient descent on Generator, using:

~~$$\min_{\theta}\mathbf{E}_{z\sim p_{\text{model}}}\big[\log\big(1-D_{\psi}(G_{\theta}(z))\big)\big]$$~~

This formula puts too much emphasis on images that are correctly classified.
Better to do gradient ascent on Generator, using:

$$\max_{\theta}\mathbf{E}_{z\sim p_{\text{model}}}\big[\log\big(D_{\psi}(G_{\theta}(z))\big)\big]$$

This puts more emphasis on the images that are wrongly classified.

Final Exam Format

- On-campus, supervised, using Inspera
- 2 hours, plus 10 minutes reading time
- Exam Format:
  - Multiple Choice Questions
  - Short Answer Questions
  - Numerical Questions
  - Graphical Questions
- Numerical and Graphical Questions similar in style to the Tutorial Questions
- Sample Exam will be provided, with a few examples of each type of Question
- Numerical answers should be correct to at least two decimal places (i.e. within 0.005 of the correct answer)
- Closed Book, no Notes or other Materials
- You may bring a UNSW approved calculator
- For Multiple Choice Questions, you should select the One Best Answer
- Negative Mark (-20%) for Multiple Choice Questions (or Sub-Questions)

Questions?