A Crash Introduction to Machine Learning with Neural Networks

Waleed Alsanie

walsanie AT kacst DOT edu DOT sa

2021

Outline

1. Machine Learning
2. Neural Networks
3. Deep Networks
4. Examples of Some Architectures

General Definition

- The ultimate task of machine learning is to simulate human learning.
- When humans observe natural phenomena, they try to find theories that explain them.
- Examples:

Observations:

{('animal','has feather', 'can fly', 'vertebrate', 'bird')
('animal','no feather', 'can't fly', 'vertebrate', 'not bird')
...,
...,
}

Theory:

Only animals with feather are birds.

Observations:

$\{(4, 16),$
$(7, 49),$
$\ldots,$
$\ldots,$
$\}$

Theory:

The second item of the tuple is the square of the first item.

General Definition

- We do not have access to the true rules T behind the observations $\mathbf{D}$, which we will refer to henceforth as the *data*.
- With some background knowledge, we need to *induce* a theory $\hat{T}$ which gives a good explanation of $\mathbf{D}$. To do so, we need to:
  - Define a space of possible theories.
  - Determine acceptance criteria.
- The set of assumptions used in defining the space of possible theories is called *inductive bias*.

A Machine Learning Problem

Given a dataset $\mathbf{D}$, an inductive bias, possibly some background knowledge, and acceptance criteria, find a theory $\hat{T}$ explaining $\mathbf{D}$ such that $\hat{T}$ satisfies the acceptance criteria.

Generalisation

- In our search for a theory, our aim is not to find one that is only consistent with the data we have seen. *Otherwise, our theory will be the data itself (we only need to memorise it)*.
- Our aim is to use this data, which we will call henceforth **training data**, to find a theory that is consistent with this data and any *future data governed by the same rules*. In other words, a theory that gives us a good **generalisation**.
- To validate our theory, we use another set called **test set** whose members were not used to find our theory and then apply our theory on it to check whether we have reached a good generalisation.

Occam's Razor

Principle -Adapted

Of two competing theories explaining some phenomena, the simpler is preferred.

Observations:

{('animal','has feather', 'can fly', 'vertebrate', 'bird')
('animal','no feather', 'can't fly', 'vertebrate', 'not bird')
. . . ,
. . . ,
}

Theory 1:
Only animals with feather are birds.

Theory 2:
Only animals with feather and can fly are birds.

Statistical Decisions

- Suppose that you have data consisting of some independent variables $\mathbf{x}$ and dependent variables $\mathbf{y}$:

$$\mathbf{y} = f(\mathbf{x})$$

- The problem can be cast as a *decision* problem where you use the values of $\mathbf{x}$ to decide about the values of $\mathbf{y}$.
- You do not have access to f, and thus you need to *estimate* it from $\mathbf{D}$, possibly with some background knowledge.

$$\mathbf{x}=\{animal, has\_feather, can\_fly, vertebrate\}$$
$$\mathbf{y}=\{is\_bird\}$$
$$\mathbf{D}=\{(yes,yes,yes,yes,bird),$$
$$(yes,no,no,yes,not\_bird),$$
$$\cdots,$$
$$\cdots,$$
$$\}$$
$$f(\mathbf{x})=\begin{cases} \text{is\_bird} = bird & \text{for} \quad \text{animal}=yes \ \& \ \text{has\_feather}=yes \\ \text{is\_bird}=not\_bird & \text{otherwise} \end{cases}$$

$$\mathbf{x}=\{x\}$$
$$\mathbf{y}=\{y\}$$
$$\mathbf{D}=\{(4,16),$$
$$(7,49),$$
$$\cdots,$$
$$\cdots,$$
$$\}$$
$$f(x)=x^2$$

Statistical Decisions

- To guide your estimation, suppose that there is a *loss* incurred when your estimated function $\hat{f}$ does not give you the correct decision.
- In this case, you are looking for an estimate $\hat{f}$ that gives you the minimum loss possible.

Given data $\mathbf{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{n}$, an inductive bias, possibly some background knowledge, and a loss function $L(\mathbf{y}, \tilde{f}(\mathbf{x}))$ which measures the amount of loss incurred by taking the decision $\tilde{f}$ given that the correct decision is $\mathbf{y}$, find a function $\hat{f}$ such that:

$$\hat{f} = \arg\min_{\tilde{f}} \; L(\mathbf{y}, \tilde{f}(\mathbf{x}))$$

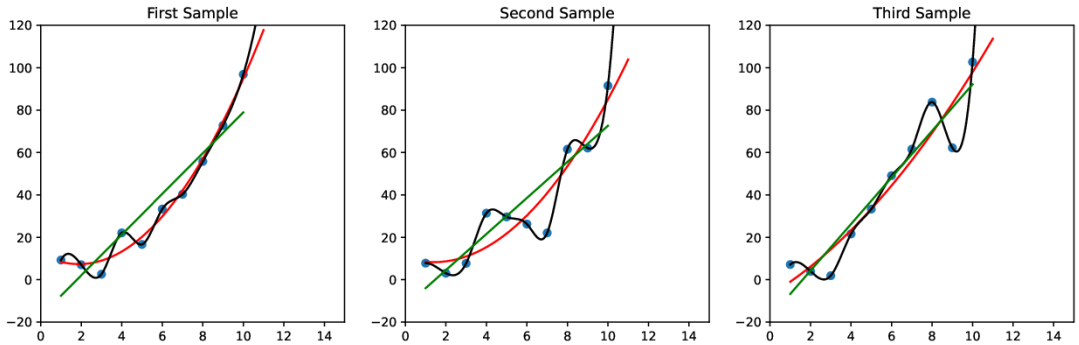
Note: The formalisation can be altered. Instead of a loss, we may define a gain we will receive when the estimated function gives the correct decision. In this case, the loss function is replaced by a *utility function* U, and the aim is to find $\hat{f}$: $\hat{f} = \arg\max_{\tilde{f}} \; U(\mathbf{y}, \tilde{f}(\mathbf{x}))$.

Loss Functions

- Examples of loss functions:

squared error loss:	$L(\mathbf{y}, \tilde{f}(\mathbf{x})) = \sum_{i=0}^{n} (\mathbf{y}_i - \tilde{f}(\mathbf{x}_i))^2$
absolute error loss:	$L(\mathbf{y}, \tilde{f}(\mathbf{x})) = \sum_{i=0}^{n} \lvert\mathbf{y}_i - \tilde{f}(\mathbf{x}_i)\rvert$
zero-one loss:	$L(\mathbf{y}, \tilde{f}(\mathbf{x})) = \sum_{i=0}^{n} \mathbb{I}[\mathbf{y}_i \neq \tilde{f}(\mathbf{x}_i)]$
Kullback-Leibler loss:	$L(p(\mathbf{y}\lvert\mathbf{x}), \tilde{p}(\mathbf{y}\lvert\mathbf{x})) = \sum_{\mathbf{x}} \sum_{\mathbf{y}} \log \left(\frac{p(\mathbf{y}\lvert\mathbf{x})}{\tilde{p}(\mathbf{y}\lvert\mathbf{x})} \right) p(\mathbf{y}\lvert\mathbf{x})$

Bias-variance Trade-off

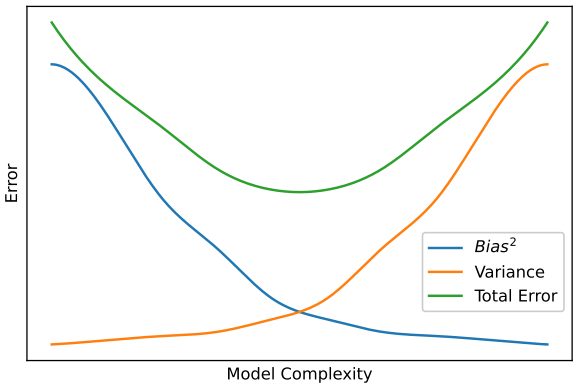


All samples were generated from $f(x) = x^2 + \epsilon$ where $\epsilon \backsim \mathcal{N}(0, 10)$

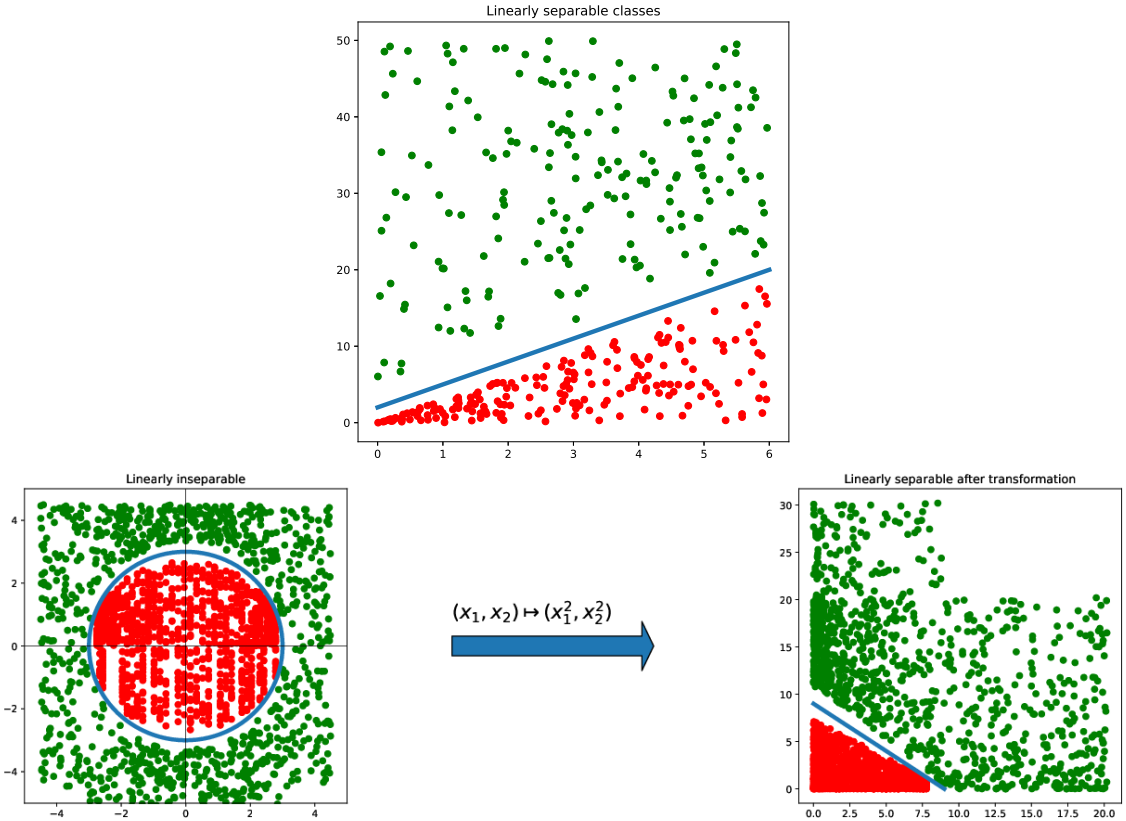
Squared Error (SE) $= (f(x) - \hat{f}(x; \mathbf{D}))^2$

$$\mathbb{E}_{\mathbf{D}}[\text{SE}] = \underbrace{(\mathbb{E}_{\mathbf{D}}[\hat{f}(x; \mathbf{D})] - f(x))^2}_{\text{Bias}^2} + \underbrace{\mathbb{E}_{\mathbf{D}}[(\mathbb{E}_{\mathbf{D}}[\hat{f}(x; \mathbf{D})] - \hat{f}(x; \mathbf{D}))^2]}_{\text{Variance}} + \text{Irreducible Error}$$

- High bias: *underfitting (over-generalisation)*
- High variance: *overfitting (over- specialisation)*
- Trade-off: *good generalisation*



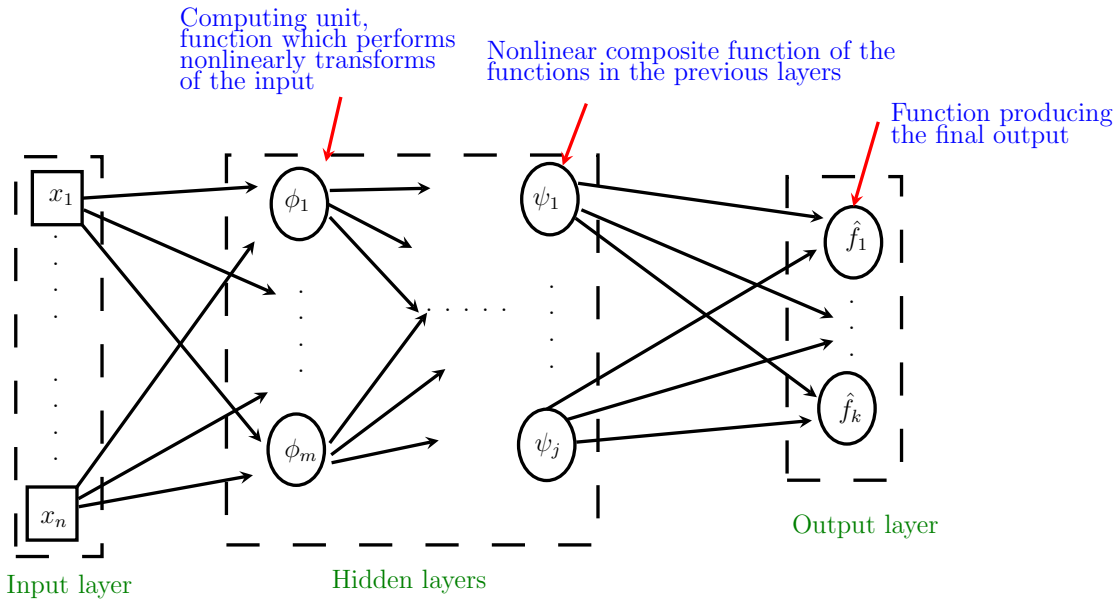
Decision Boundary and Linear Separability



The instance are not linearly separable in the original space but linearly separable in another space, which we may transform the instances to:

$$\hat{y} = \hat{f}(\phi(x_1, x_2))$$

What it is

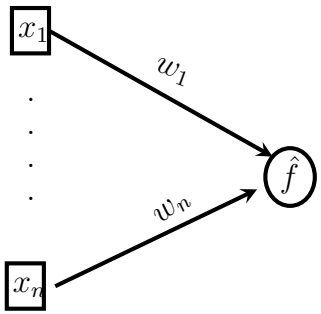


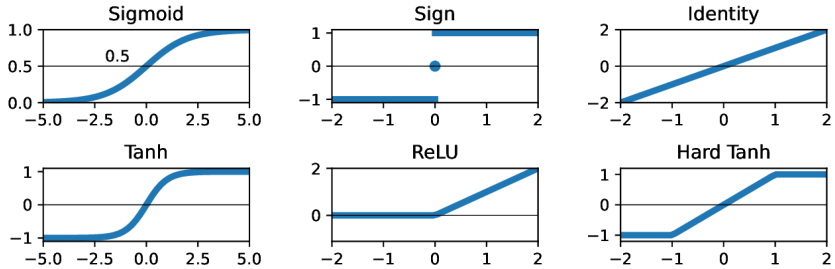
- A *neural network (NN)* is a composite function whose structure was inspired by a simple form of the *human nervous system*:

$$\hat{y}_i = \hat{f}_i(\psi_1(\ldots\phi_1(x_1,\ldots,x_n),\ldots,\phi_m(x_1,\ldots,x_n)\ldots),\ldots,$$
$$\psi_j(\ldots\phi_1(x_1,\ldots,x_n),\ldots,\phi_m(x_1,\ldots,x_n)\ldots))$$

- Computing units, known as *neurons*, are functions, referred to as *activation functions*. Neurons in the *hidden layers* perform nonlinear transformation of their input. Neurons at the output layer act as decision functions to produce the output.

Perceptron

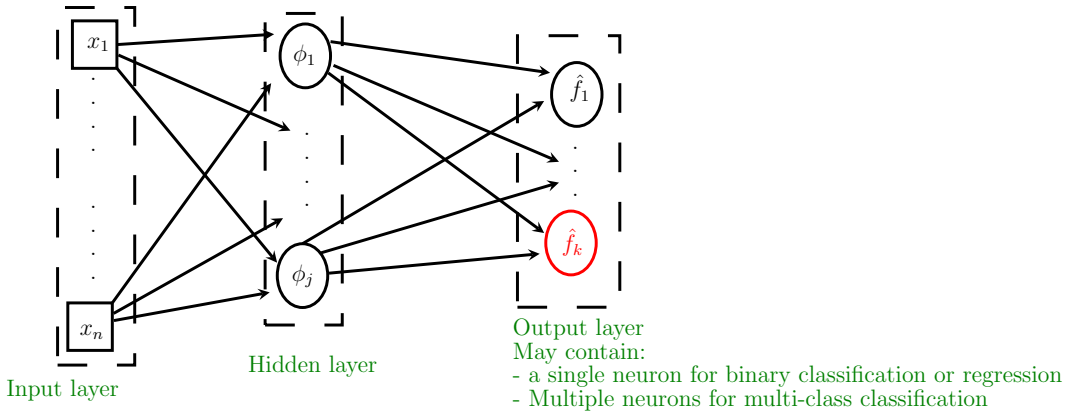




- The activation function, acts on a linear combination of the input: $\hat{y} = \hat{f}(\sum_{i=1}^{n} x_i w_i)$

Sigmoid	$f(z) = \frac{1}{1+e^{-z}}$	Classification (logistic regression)
Sign	$f(z) = \begin{cases} 1 & \text{for} \quad z > 0 \\ 0 & \text{for} \quad z = 0 \\ -1 & \text{for} \quad z < 0 \end{cases}$	Classification
Identity	$f(z) = z$	Regression (linear in this case) and classification
Hyperbolic Tangent (Tanh)	$f(z) = \frac{e^{2z}-1}{e^{2z}+1}$	Classification
Rectified Linear Unit (ReLU)	$f(z) = \max(0, z)$	Classification
Hard Tanh	$f(z) = \max(\min(1, z), -1)$	Classification

Single Hidden Layer Feedforward



- Neurons in the hidden layer perform nonlinear transformation of the input (*hidden representation*).
- Neurons at the output layer produce the output based on the hidden representation. The output layer may contain:
  - A single neuron of an activation function for binary classification or regression.
  - Multiple neurons for multi-class classification:
    - *One-vs-all*: binary classification activation function at each neuron, e.g. sigmoid.
    - *Softmax*: a probability distribution over the classes. Assuming there are k classes:

$$\text{Softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}, \qquad \text{where } z_i \text{ is the } i\text{th component of the vector } \mathbf{z}$$

- It may approximate any function, depending on the number of neurons in the hidden layer (*universal approximation*).

Learning

Assuming the structure and the activation functions are set a priori (inductive bias), learning is left with estimating the paramters $\mathbf{W}$ such that, for some loss function L:

$$\hat{\mathbf{W}} = \arg\min_{\tilde{W}} L(\mathbf{y}, \hat{f}(\mathbf{x}; \tilde{W}))$$

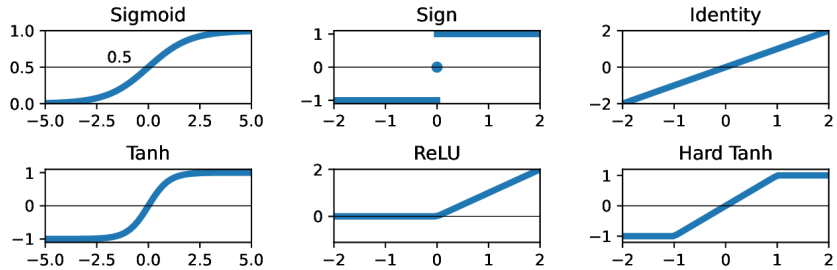
Examples of some loss functions and how they are used with some activation functions (for a single training instance):

Loss	Form	Task	y	Activation Func. in Output
Squared error loss	$(y - \hat{f}(\mathbf{x}))^2$	regression	real value	identity
Hinge Loss	$\max(0, 1 - y\hat{f}(\mathbf{x}))$	binary classification	$\{-1, 1\}$	identity
Cross-entropy	$-\sum_{i=1}^{k} y_i \log(\hat{f}_i(\mathbf{x}))$	classification (k classes)	$y_i = \begin{cases} 1 & i \text{ is the true class} \\ 0 & \text{otherwise} \end{cases}$	sigmoid & softmax

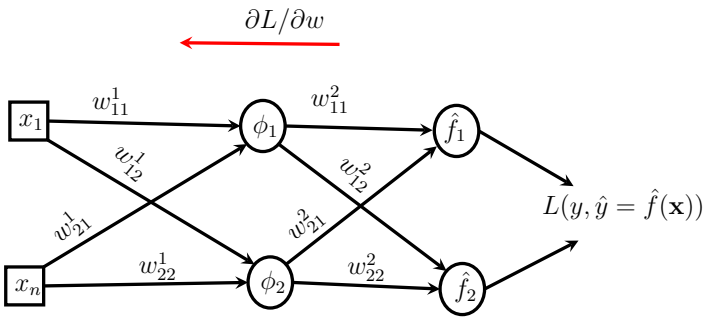
In principle, any combination of output activation functions and loss functions can be used as long as they make sense in the application.

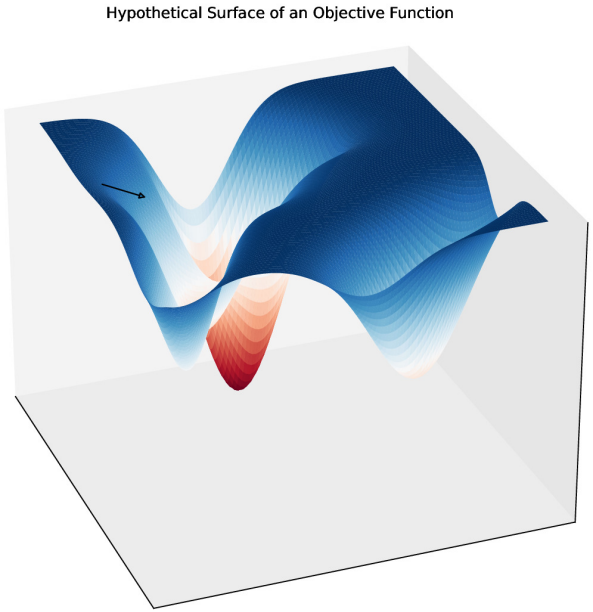
Note:

- ReLU and (Hard) Tanh are not very common in the output layer.
- Sign function is unfavourable due to its differentiability.



Learning -Gradient Descent (GD) with Backpropagation (BP)





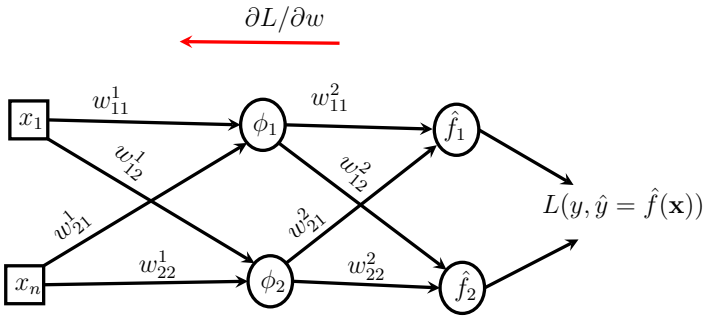
- Initialise the weights w in $\mathbf{W}$.
- For each *epoch* (a cycle in which the whole instance are processed):

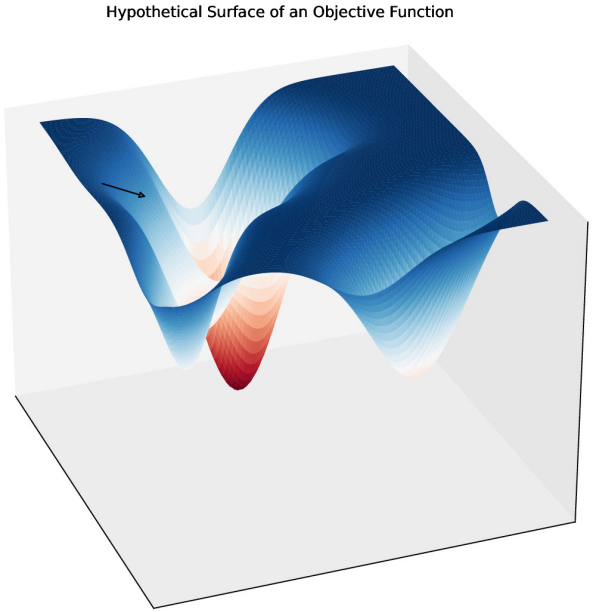
1. Feed the all the instances to the network and compute the loss $L = \sum_{i=1}^{n} L_i$
2. For every w, compute $\frac{\partial L}{\partial w}$ starting from the output layer moving *backward* to the input layer.
3. Update the weights by moving in the opposite direction of the gradient using a learning rate η:

$$w = w - \eta \frac{\partial L}{\partial w}$$

Note: Slow when the dataset is large, needs enough memory to process the whole data at once.

Learning -Stochastic Gradient Descent (SGD) with BP



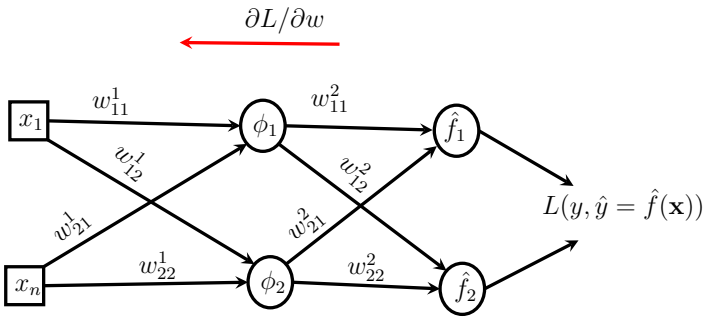


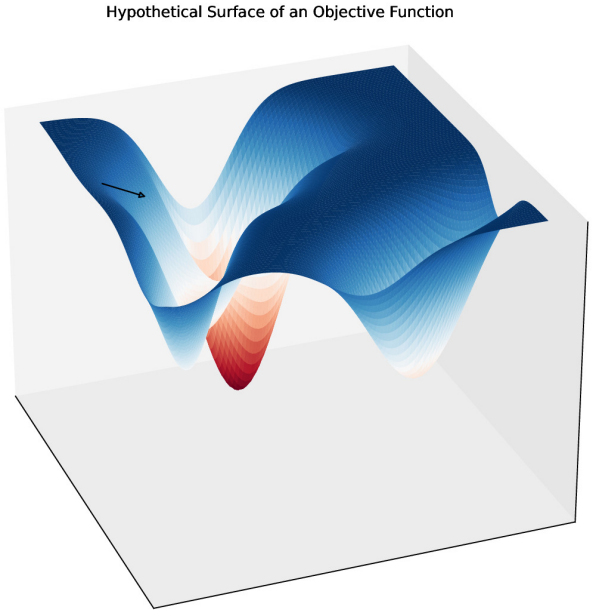
- Initialise the weights w in $\mathbf{W}$.
- For each *epoch*:
  - For the number of instances, pick a random instance that has not been processed in this epoch:
    1. Feed it to the network and compute the loss L
    2. For every w, compute $\frac{\partial L}{\partial w}$ starting from the output layer moving *backward* to the input layer.
    3. Update the weights by moving in the opposite direction of the gradient using a learning rate η:

$$w = w - \eta \frac{\partial L}{\partial w}$$

Note: Weights are updated with respect to the loss of one instance only, this may lead to poor learning.

Learning -Mini-batch Stochastic Gradient Descent with BP

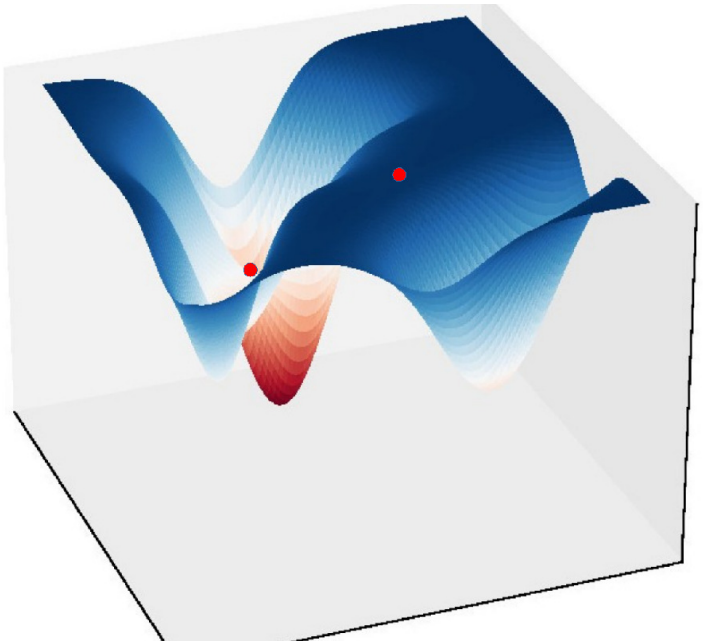




- Initialise the weights w in $\mathbf{W}$.
- Divide the training set into m batches.
- For each *epoch*:
  - For each batch $\mathbf{B}$:
    1. Feed it to the network and compute its loss $L = \sum_{\mathbf{x} \in \mathbf{B}} L_{\mathbf{x}}$
    2. For every w, compute $\frac{\partial L}{\partial w}$ starting from the output layer moving *backward* to the input layer.
    3. Update the weights by moving in the opposite direction of the gradient using a learning rate η:
$$w = w - \eta \frac{\partial L}{\partial w}$$

A trade off between GD and SGD. The size of the mini-batch needs to be set carefully (considering the memory). It is a common practice to set the size as a power of 2. Common values are $32, 64, 128$ and 256.

Learning -Note



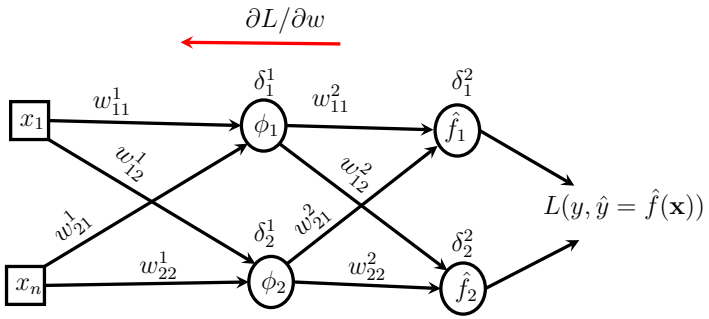
Gradient descent finds a local minimum. It might be trapped into a saddle point or a plateau.

Learning -Dynamic Programming in BP

Chain rule of differentiation:

$$\frac{df(g(h(x)))}{dx} = \frac{df(g(h(x)))}{dg(h(x))}\frac{dg(h(x))}{dh(x)}\frac{dh(x)}{x}$$

Let: $v_1 = w_{11}^1 x_1 + w_{21}^1 x_2$, $v_2 = w_{12}^1 x_1 + w_{22}^1 x_2$,
$a_1 = \phi_1(v_1)$, $a_2 = \phi_2(v_2)$,
$z_1 = a_1 w_{11}^2 + a_2 w_{21}^2$, $z_2 = a_1 w_{12}^2 + a_2 w_{22}^2$,
$y_1 = f_1(z_1)$ and $y_2 = f_2(z_2)$



$$\frac{\partial L}{\partial w_{11}^2} = \overbrace{\frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial z_1}}^{\delta_1^2}\frac{\partial z_1}{\partial w_{11}^2} \qquad \frac{\partial L}{\partial w_{11}^1} = \overbrace{\frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial z_1}}^{\delta_1^2}\frac{\partial z_1}{\partial a_1}\frac{\partial a_1}{\partial v_1}\frac{\partial v_1}{\partial w_{11}^1} + \overbrace{\frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial z_2}}^{\delta_2^2}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial v_1}\frac{\partial v_1}{\partial w_{11}^1}$$

$$\frac{\partial L}{\partial w_{21}^2} = \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial z_1}\frac{\partial z_1}{\partial w_{21}^2} \qquad \frac{\partial L}{\partial w_{21}^1} = \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial z_1}\frac{\partial z_1}{\partial a_1}\frac{\partial a_1}{\partial v_1}\frac{\partial v_1}{\partial w_{21}^1} + \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial z_2}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial v_1}\frac{\partial v_1}{\partial w_{21}^1}$$

re-write (assuming w_{11}^1 is considered first in this layer):

re-write (assuming w_{11}^2 is considered first):

$$\frac{\partial L}{\partial w_{21}^2} = \delta_1^2\frac{\partial z_1}{\partial w_{21}^2}$$

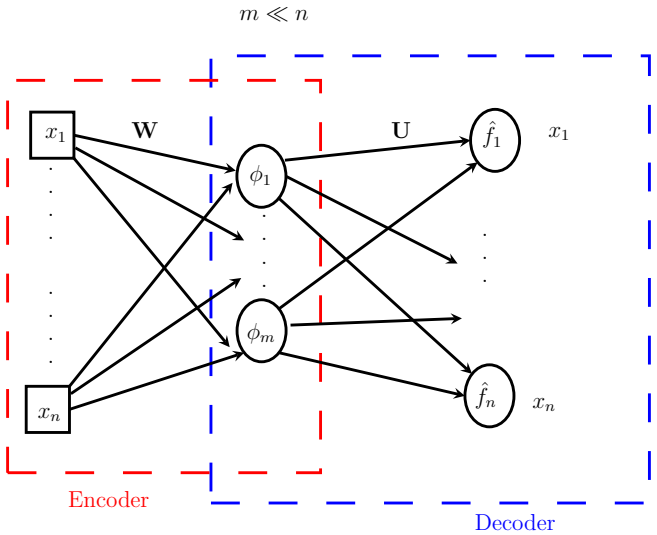
$$\frac{\partial L}{\partial w_{11}^1} = \delta_1^2\frac{\partial z_1}{\partial a_1}\frac{\partial a_1}{\partial v_1}\frac{\partial v_1}{\partial w_{11}^1} + \delta_2^2\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial v_1}\frac{\partial v_1}{\partial w_{11}^1} = \overbrace{(\delta_1^2\frac{\partial z_1}{\partial a_1}\frac{\partial a_1}{\partial v_1} + \delta_2^2\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial v_1})}^{\delta_1^1}\frac{\partial v_1}{\partial w_{11}^1}$$

$$\frac{\partial L}{\partial w_{21}^1} = \delta_1^1\frac{\partial v_1}{\partial w_{21}^1}$$

Likewise for the other weights

Autoencoder

Dimensionality reduction:



Linear:

$\phi = \sum_{i=1}^{n} x_{ji} w_{im}$

- Let the data be $\mathbf{D}_{j \times n}$ (j instances each of n dimensions), $\mathbf{\Phi}_{j \times m} = \mathbf{D}_{j \times n} \mathbf{W}_{n \times m}$
- $\mathbf{\Phi}$ is a reduced (latent) representation of $\mathbf{D}$
- The original data is approximated (reconstructed) as: $\mathbf{D}_{j \times n} \approx \mathbf{\Phi}_{j \times m} \mathbf{U}_{m \times n}$

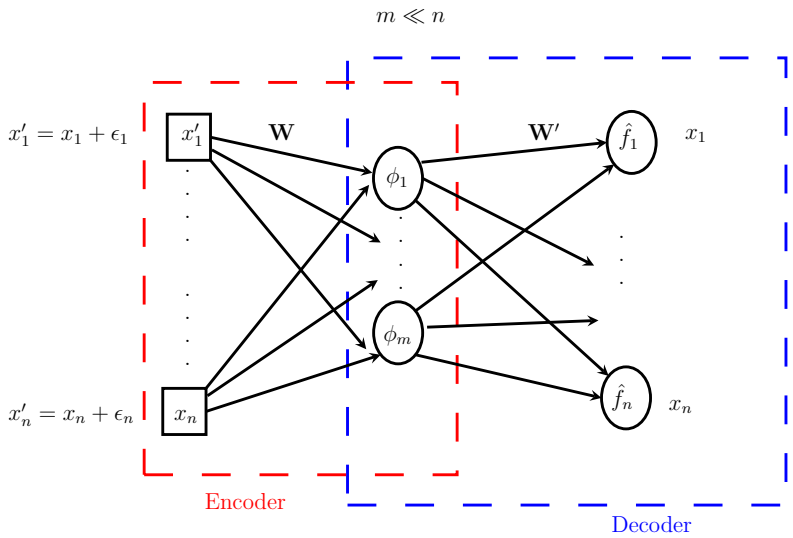
$$\text{Let } \mathbf{\Phi U} = \tilde{\mathbf{D}}$$

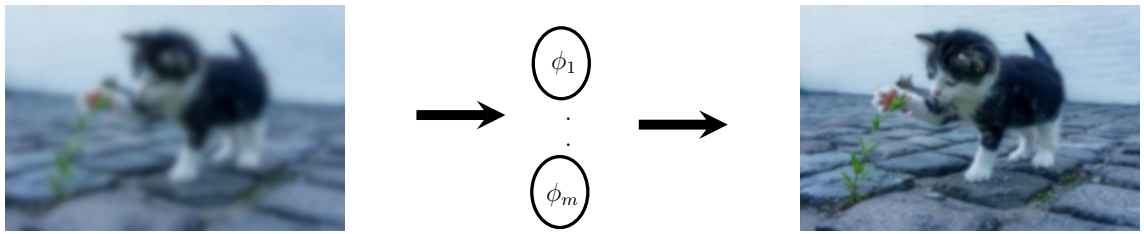
$$\text{Backpropagation with } L = \|\mathbf{D} - \tilde{\mathbf{D}}\|_F^2 = \sum_{i=1}^{j} \sum_{k=1}^{n} (d_{ik} - \tilde{d}_{ik})^2 \text{ (squared Frobenius norm)}$$

* Nonlinear reduced representation can also be achieved by using sigmoid in the hidden units.
* *Sparse autoencoder*: $m \gg n$

Denoising Autoencoder

Add noise $\epsilon \backsim P$ to the data according to some distribution P and train to reconstruct the original data.



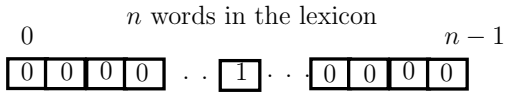


Word Embedding (word2vec)

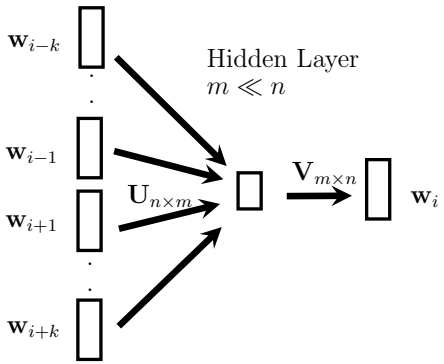
Distributional Semantics Hypothesis

Words that occur in the same contexts tend to convey similar meanings.

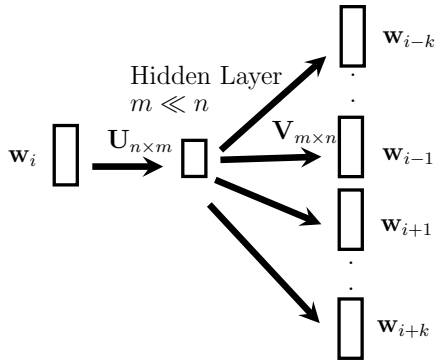
Let the words w be represented in *one-hot encoding*:



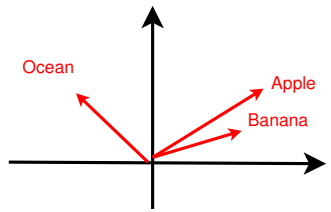
Continuous bag-of-words (CBOW):



Continuous skip-gram:



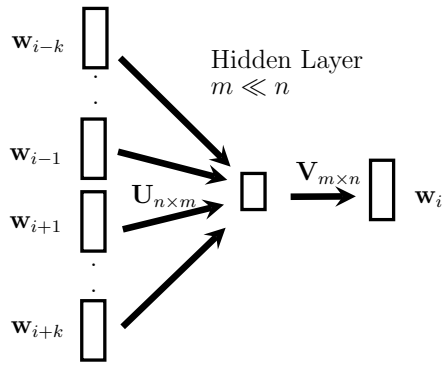
$\mathbf{U}_{n\times m}$ is the word embedding (latent representation) of $\mathbf{w}_i$ in the m dimensional semantic space.



Simple exercise: Think of a product recommendation system in e-commerce using word2vec.

Word Embedding (word2vec) -Notes

Continuous bag-of-words (CBOW):

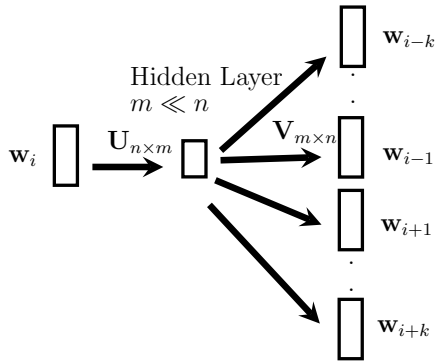


- Faster to train.
- Context windows are usually in the range $5 - 10$.
- Performed better in capturing (morpho-)syntactically related words, e.g.: small, smaller, smallest, etc.
- Word order in the context is not considered, thus the name bag-of-words (the output of the hidden layer is a weighted sum of the context words):

$$\forall c \in \{1, \ldots, m\}, \quad \phi_c = \sum_{i=1}^{2k} \sum_{j=1}^{n} x_{ij} u_{jc}$$

- Suitable for small datasets.

Continuous skip-gram:

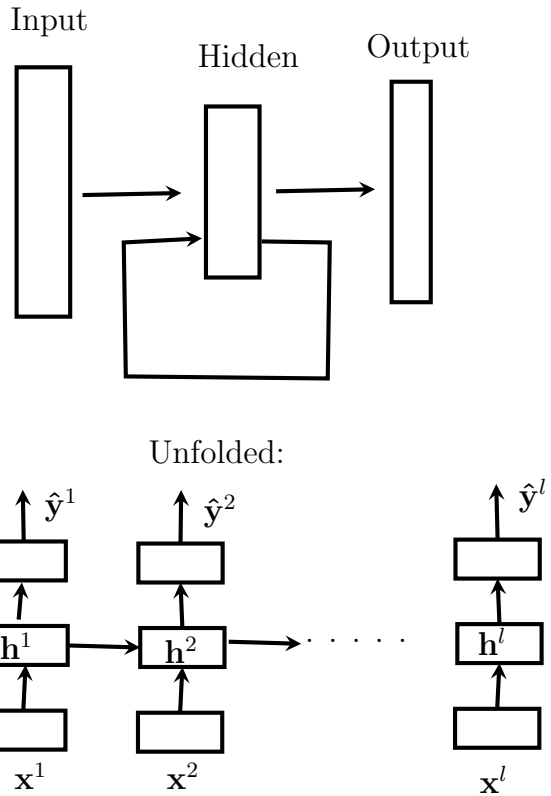


- Slower to train.
- Long context windows can be used for training.
- Performed significantly better in capturing semantically related words, e.g.: small, large, medium.
- Word order matters. Long distance words are weighed less (sampled less during training).
- Better at capturing infrequent words, and when trained on large datasets.

$\mathbf{U}_{n\times m}$ in CBOW and $\mathbf{V}_{m\times n}$ in Skip-gram are shared amongst the context words.

Recurrent NN (RNN)

Suitable for processing sequences: time series, texts, speech, etc.



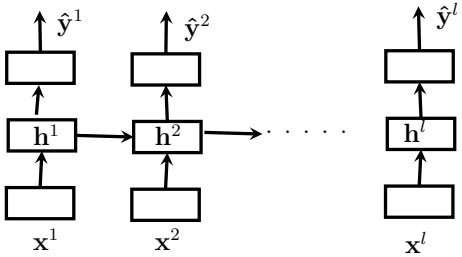
Decision on $\mathbf{y}^t$ depends on $\mathbf{x}^t$ and the hidden representation of the sequence up to $t-1$, which is $\mathbf{h}^{1:t-1}$.

Extra weight matrix $\mathbf{W}_h$ (hidden-to-hidden weight matrix)

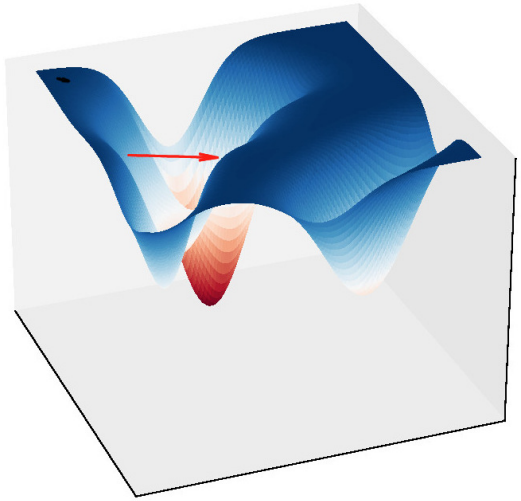
Vanishing and Exploding Gradients

Chain rule of differentiation:

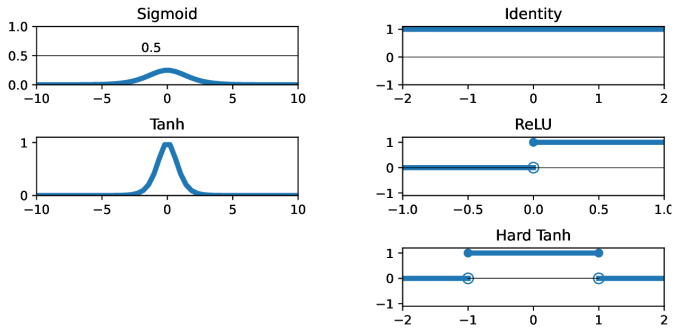
$$\frac{df(g(\ldots h(x)\ldots))}{dx} = \frac{df(g(\ldots))}{dg(\ldots)} \cdots \frac{dh(x)}{x}$$



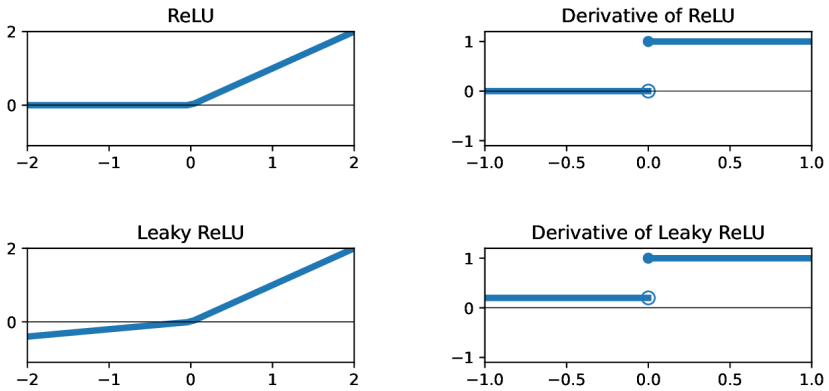
Training an RNN may require backpropagating through long sequences.



- *Vanishing gradient*: the gradient becomes very small ≈ 0 through long backpropagation paths.
  - ReLU and Hard Tanh in the hidden neurons of deep networks (ReLU is more common) instead of Sigmoid and Tanh.
- *Exploding gradient*: the gradient becomes large through long backpropagation paths.
- Some proposed solutions:
  - Gradient clipping: e.g.: set a minimum/maximum threshold on the gradient's value.
  - Other architectures: Long Short Term Memory, Gated Recurrent Unit (GRU).



ReLU and Leaky ReLU



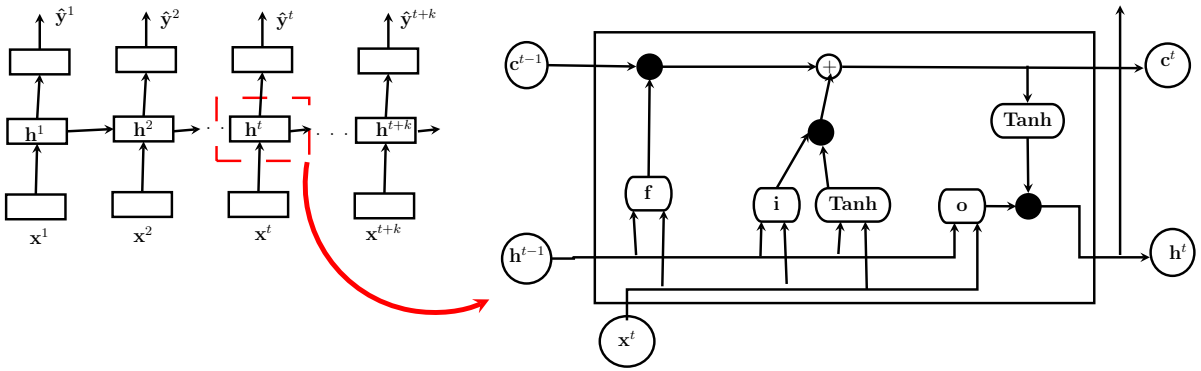
- The derivative of ReLU is 0 on the negative line.
- This might turn the neuron down, a situation known as *dead neuron*. e.g.:
  - The input to this neuron is always positive and the weights going to this neuron reach negative values or the vice versa.
- The weights will not be updated during backpropagation, for the updating term is always 0:

$$w = w - \eta \frac{\partial L}{\partial w}$$

- *Leaky ReLU* was proposed to avoid this problem. For some $\alpha \in (0, 1)$, it is defined as:

$$\begin{cases} \alpha x & \text{for} \quad x < 0 \\ x & \text{for} \quad \text{otherwise} \end{cases}$$

Long Short Term Memory (LSTM) NN

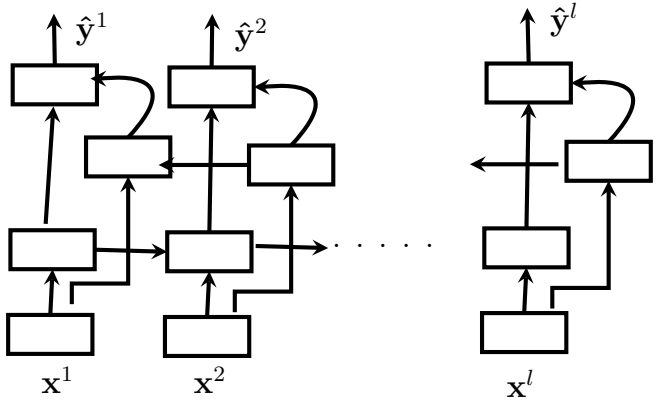


$$\mathbf{f}=\textbf{sigmoid}(\mathbf{W}_f\mathbf{x}^t+\mathbf{U}_f\mathbf{h}^{t-1}) \quad \mathbf{i}=\textbf{sigmoid}(\mathbf{W}_i\mathbf{x}^t+\mathbf{U}_i\mathbf{h}^{t-1}) \quad \mathbf{o}=\textbf{sigmoid}(\mathbf{W}_o\mathbf{x}_t+\mathbf{U}_o\mathbf{h}^{t-1})$$
$$\tilde{\mathbf{c}}^t=\textbf{tanh}(\mathbf{W}_c\mathbf{x}^t+\mathbf{U}_c\mathbf{h}^{t-1}) \quad \mathbf{c}^t=\mathbf{f}\odot\mathbf{c}^{t-1}+\tilde{\mathbf{c}}^t\odot\mathbf{i} \quad \mathbf{h}^t=\mathbf{o}\odot\textbf{tanh}(\mathbf{c}^t)$$

$$\frac{\partial\mathbf{c}^t}{\partial\mathbf{c}^{t-1}}=\mathbf{f}+\frac{\partial\mathbf{f}}{\partial\mathbf{c}^{t-1}}\mathbf{c}^{t-1}+\frac{\partial\tilde{\mathbf{c}}^t}{\partial\mathbf{c}^{t-1}}\mathbf{i}+\frac{\partial\mathbf{i}}{\partial\mathbf{c}^{t-1}}\tilde{\mathbf{c}}^t$$

- The *cell state* $\mathbf{c}^{t-1}$ was introduced to carry the information amongst the states (long term memory). The following gates were also introduced to control the flow of the information:
  - $\mathbf{f}$ (*forget gate*): decides which information in $\mathbf{c}^{t-1}$ is important, and thus needs to carry on the sequence with the light of the current time step.
  - $\mathbf{i}$ (*input gate*): decides which information from the current time step needs to be added to $\mathbf{c}^{t-1}$.
  - $\mathbf{o}$ (*output gate*): decides which information in $\mathbf{c}^t$ needs to be passed to the hidden state after this time step.
- The **tanh** vectors hold the information that needs to be passed and the **sigmoid** vectors decide which information is important.

Bi-LSTM



[The presentation is about machine learning. It covers the main concepts in machine learning and neural networks. Neural networks have become popular with the availability of powerful computers. Their popularity can be noticed with the amount of published articles.]

- Refers to **Bi**directional **LSTM**.
- Very popular in the NLP community.
- Useful when you need to consider the hidden states of the preceding and succeeding sequences.

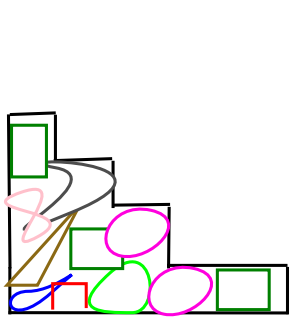
Further Topics

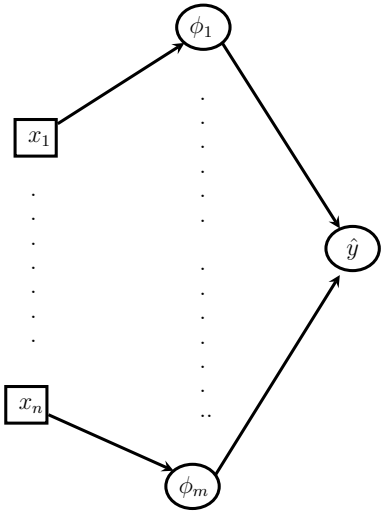
- Some important topics which were not covered:
  - *Convolutional neural networks*: popular in the computer vision community. Its architecture was inspired by the visual cortex of the cats.
  - Finding good settings of the learning rate in:

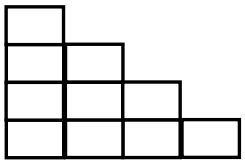
$$w = w - \eta \frac{\partial L}{\partial w}$$

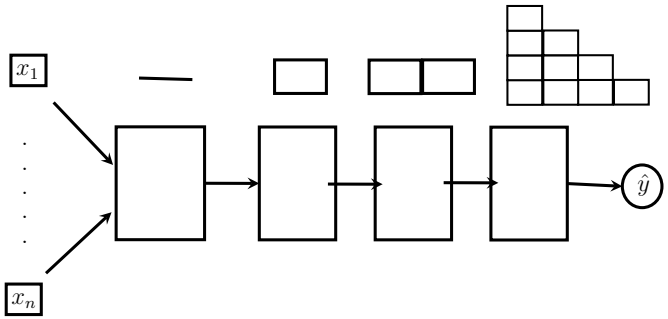
to avoid wiggling around the minimum in weight updating. e.g. *momentum techniques*.

Rationale









- Multiple hidden layers.
- Utilises function composition to reveal the composite structure of the data, and thus reduce the number of neurons (reducing model complexity).
- Modularity: complex networks solving complex problems might be built by a pipeline of sub-networks each of which solves a subproblem.

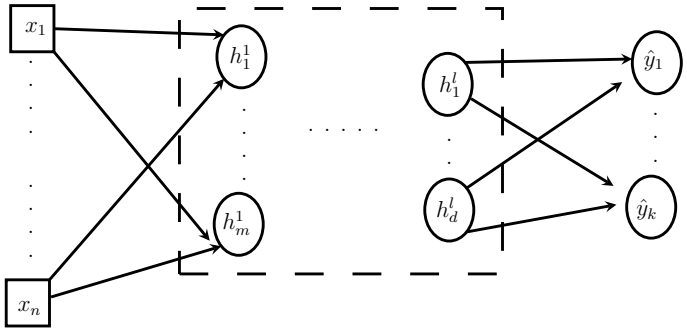
[. . . [[[morpheme] word + syntactic info. + semantic info.] syntactic structure] . . .]

Challenges

- Vanishing and exploding gradients exist.
- Very deep networks are complex models with very high numbers of parameters (in millions in many cases). They need very large datasets to train.
- High chance of overfitting.
- Very deep networks need high computational power to train. GPUs are normally needed to speed up the training (intensive matrix multiplications).

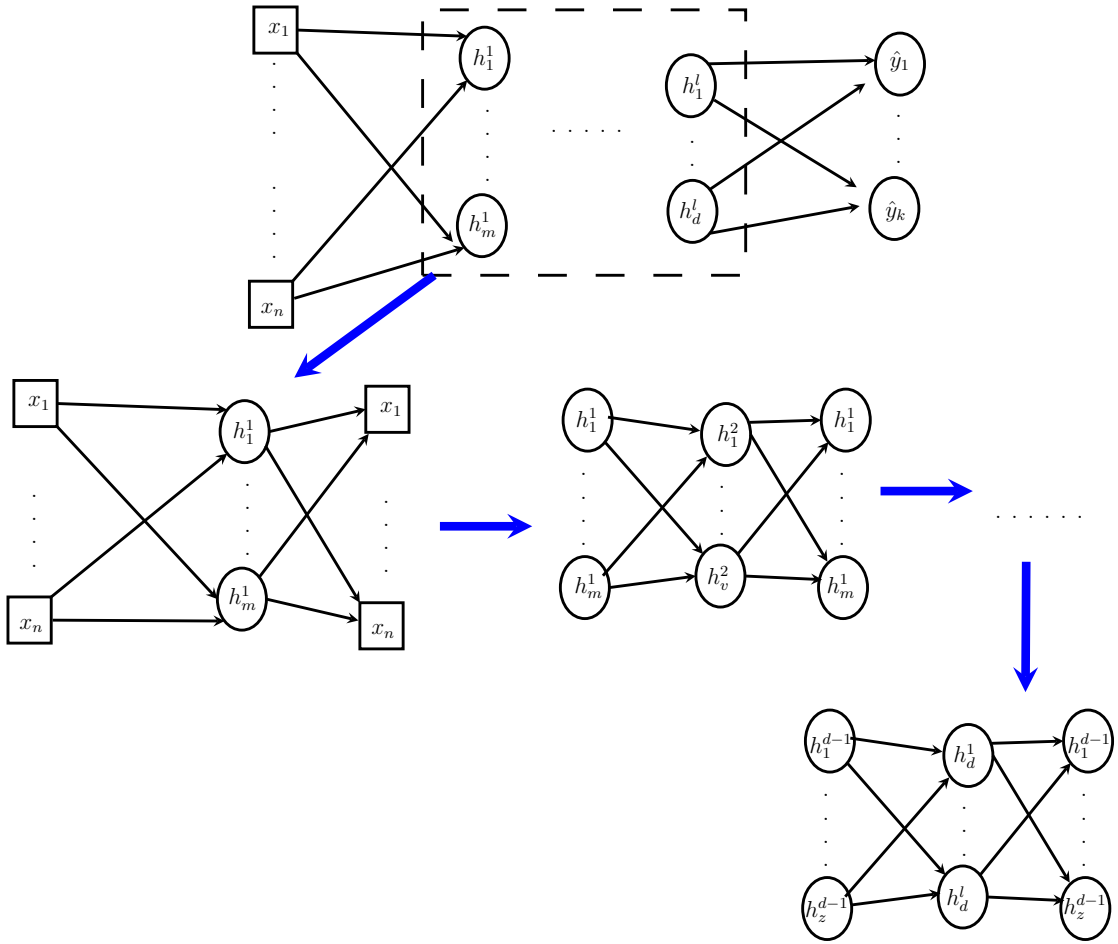
Pretraining

Train the hidden layers separately.



- Good initial points for the backpropagation of the intended task.
- Mitigating the effects of vanishing gradient on the final performance.
- *Transfer learning*: training a multi-purpose architecture. i.e. train the hidden layers to process the main elements in structured data, and then fine-tune them according to the intended tasks. e.g.:
  - Pretrain the layers to embed word segments, words, sentence boundary, etc.
  - For a specific task, run backpropagation on a dataset to fine-tune the parameters: question-answering, machine translation, etc.
- Reducing the chance of overfitting: when gradients vanish, the parameters of the late layers might be adapted to minimise the loss irrespective of the parameters of the early layers.

Unsupervised Pretraining



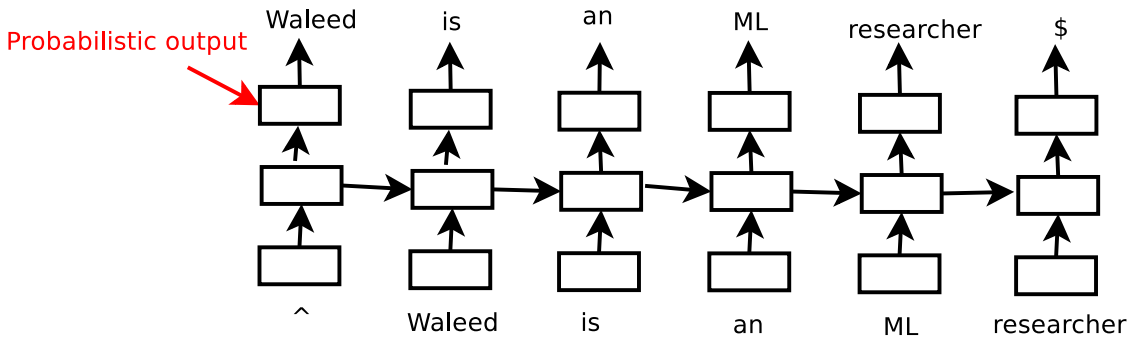
Train each layer, or each group of hidden layers, as an autoencoder, or a multi-layer autoencoder.

Avoiding Overfitting

Some techniques:

- *Regularisation*: add a complexity penalty term to the loss function, e.g. in L_1-regularisation: $\lambda \sum_{i=1}^{d} |w_i|$ is added and in L_2-regularisation: $\lambda \sum_{i=1}^{d} w_i^2$ is added. λ is a factor controlling the penalisation amount.
- *Bagging* and *subsampling*:
  - Sample m sets from the base training set, sample with replacement in bagging (normally with the same size of the base training set) and without replacement in subsampling (in this case the size of each set needs to be less than the size of the base training set).
  - Train m networks each with one of the sampled sets. In inference, use each network to generate an output and report the average.
- *Dropout*:
  - From the base network, generate m sub-networks by randomly dropping out nodes from the input layer with some probability and from the hidden layers with a different probability. Probabilities are conventionally in the range $[0.2 - 0.5]$
  - Train each network with a mini-batch. In inference, use each network to generate an output and report the average.
- *Early stopping*: The network is not trained until convergence. Instead:
  - A validation set is created and the network is trained on the training set and applied to the validation set.
  - When the error starts to rise on the validation set, the training is stopped even if it is still declining in the training set.

Language Modelling

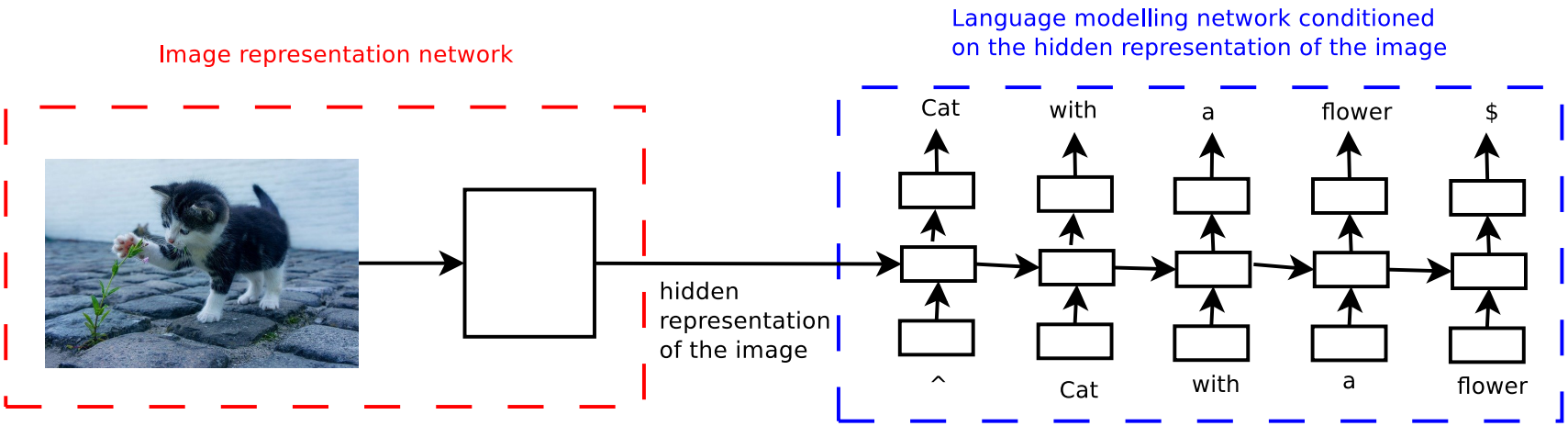


- Input at time t: one-hot encoding of the word.
- Output at time t: probability distribution over the lexicon words.
- The model can be used to generate random texts:
  - Type the start symbol '^'.
  - The model will pick a random word according to the distribution resulting from the training. It will use this word as input to the next time step and it will continue the process until it generates the end mark '$'.

Examples:

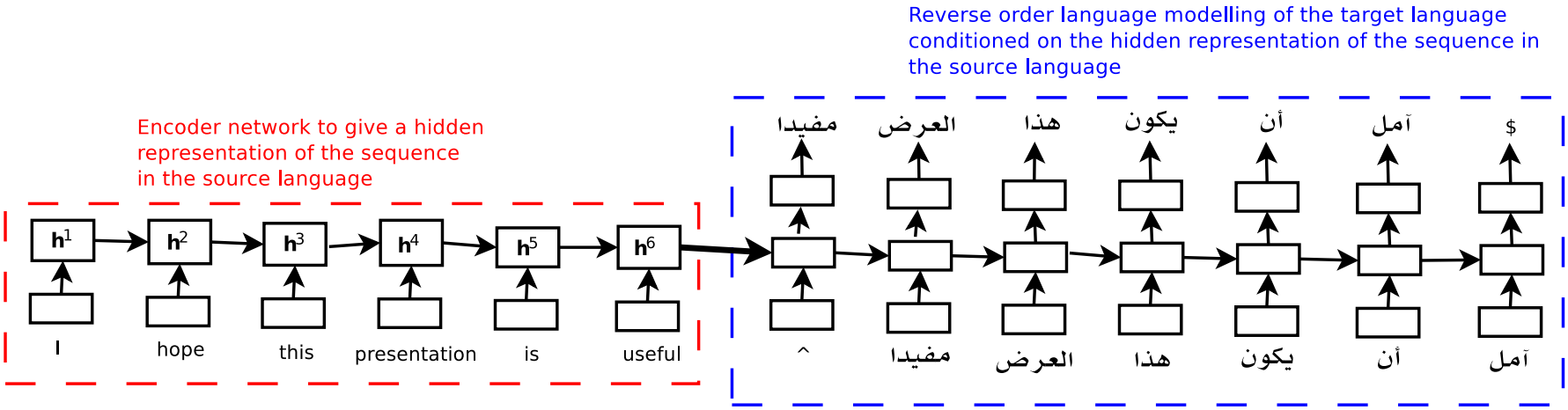
{'^Waleed is a movie$', '^LaTeX is a typesetting software$', '^No I don't$', . . . }

Image Captioning



The image representation network can be any network, e.g.: a convolutional neural network.

Machine Translation

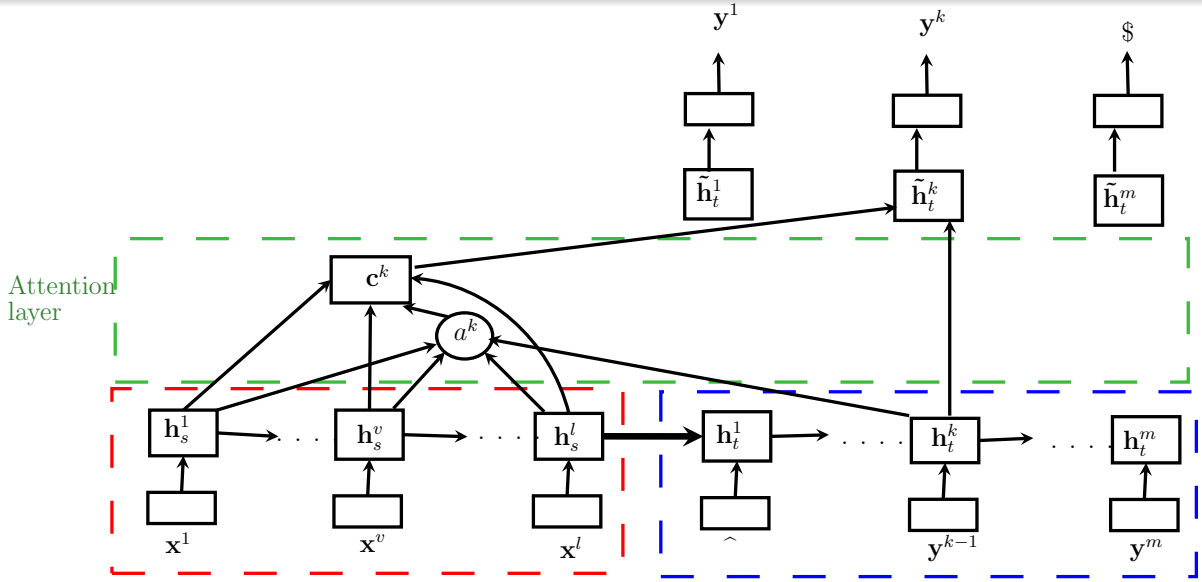


Sequence-to-sequence model.

The language model can be in the normal order. However, training with a reverse order language model makes less the distance between the aligned words in the source and target languages.

* This model is referred to in the literature as the *encoder-decoder model*. The set of the hidden states of the source sequence, $\mathbf{h}^1, \ldots, \mathbf{h}^6$ above, is referred to as the *memory*.

Attention-based Models



- Some words in the source sequence may play more important roles than the others in generating an output y^k in the target sequence. The model needs to be trained to *attend* to these words.
- A *context* vector for each target word is added to control the source words that need to be attended. Different attention mechanisms have been proposed, e.g: a linear combination of the source states[1]:

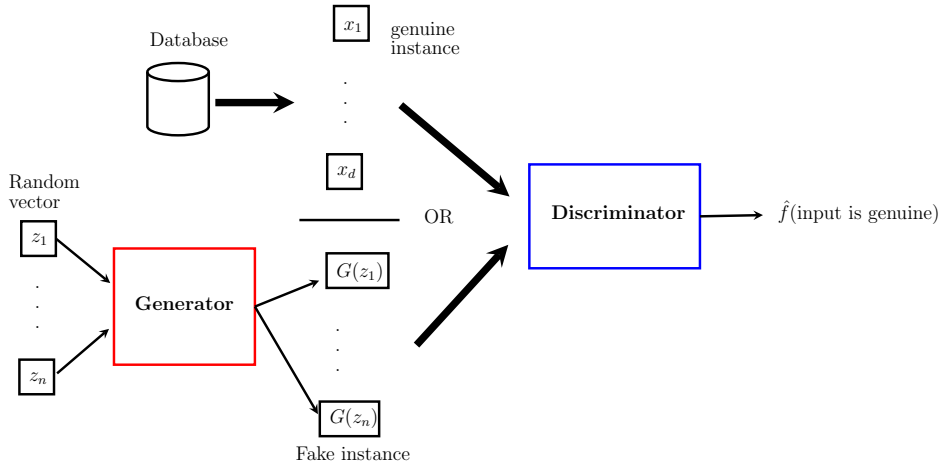
$$c^k = \sum_{j=1}^{l} a(k,j)\mathbf{h}_s^j$$

$a(k,i)$ is called *attention variable*. It measures the importance of the source word i to the target word k. It is the ith component of the softmax of the dot product (similarity) between the source state vectors $\mathbf{h}_s$ and the target state vector $\mathbf{h}_t^k$:

$$\text{Softmax}(\mathbf{h}_s \cdot \mathbf{h}_t^k)_i = a(k,i) = \frac{e^{\mathbf{h}_s^i \cdot \mathbf{h}_t^k}}{\sum_{j=1}^{l} e^{\mathbf{h}_s^j \cdot h_t^k}}$$

[1] Luong, M., Pham, H., and Manning, C.D. (2015). 'Effective Approaches to Attention-based Neural Machine Translation', *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*.

Generative Adversarial Networks (GANs)



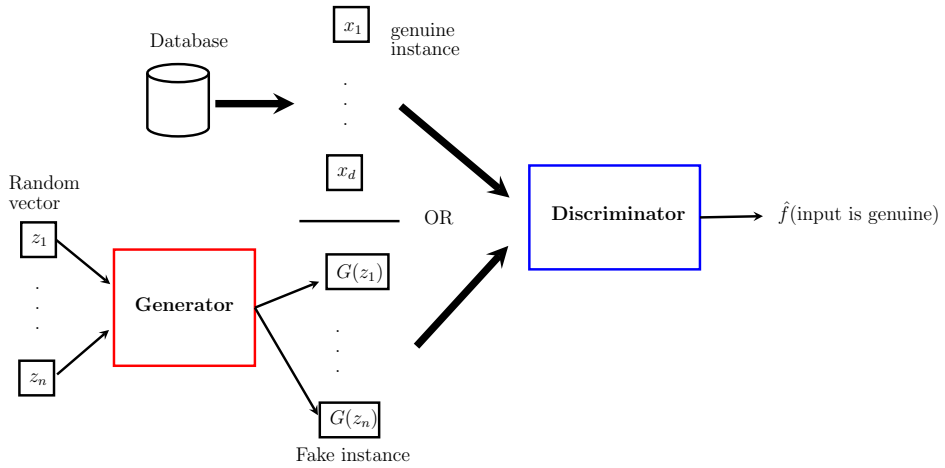
- Composed of two sub-networks, *generator* and *discriminator*, which play the following game during the training:
  - The generator takes random vectors $\{\mathbf{z}_1, \ldots, \mathbf{z}_m\}, \mathbf{z}_i \sim P$, generates instances $\{G(\mathbf{z}_1), \ldots, G(\mathbf{z}_m)\}$ and passes them to the discriminator to deceive it.
  - The discriminator is fed a set of inputs, $\{\mathbf{x}_i\}_{i=1}^m \bigcup \{G(\mathbf{z}_i)\}_{i=1}^m$, consisting of *genuine* instances, $\{\mathbf{x}_i\}_{i=1}^m$, and *generated* instances, $\{G(\mathbf{z}_i)\}_{i=1}^m$. For each instance it reports its confidence on whether it is genuine, $\hat{f}$.
  - The discriminator adapts to maximise its ability to discriminate the input:

$$U = \mathbb{E}_{\mathbf{x}}[\log(\hat{f}(\mathbf{x}))] + \mathbb{E}_{\mathbf{z}}[\log(1 - \hat{f}(G(\mathbf{z})))]$$

  - The generator adapts to minimise the discriminator's ability to discriminate the inputs:

$$L = \mathbb{E}_{\mathbf{z}}[\log(1 - \hat{f}(G(\mathbf{z})))]$$

Generative Adversarial Networks (GANs)



- The aim is to solve the following minimax problem:

$$\min_G \max_{\hat{f}} V(G, \hat{f}) = \mathbb{E}_{\mathbf{x}}[\log(\hat{f}(\mathbf{x}))] + \mathbb{E}_{\mathbf{z}}[\log(1 - \hat{f}(G(\mathbf{z})))]$$

- **For** i iterations:
  - Repeat k times:
    - Use the generator to generate m instances and sample m genuine instances. Feed these data to the discriminator, compute U and use backpropagation through the discriminator to adapt its weights to maximise U (gradient ascent).
  - Use the generator to generate m instances. Feed the instances to the discriminator, compute L and use backpropagation through the two networks to adapt the generator's weights to minimise L (gradient descent).

Epilogue

Some issues:

- Complexity.
- Interpretability.
- Relational Learning.
- Unsupervised Learning.

A good read:

Marcus, G. (2018). 'Deep Learning: A Critical Appraisal', *arXiv:1801.00631 [cs, stat]*

The code used to generate the plots in this presentation can be found at:
`https://github.com/walsanie/ML_NN_Pres_Plots.git`

Q & A

Q & A