Imperial College London

NOTES

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Deep Learning

Author:
W (CID: your college-id number)

Date: February 27, 2020

1 Deep Feed Forward Networks

1.1 Perceptron

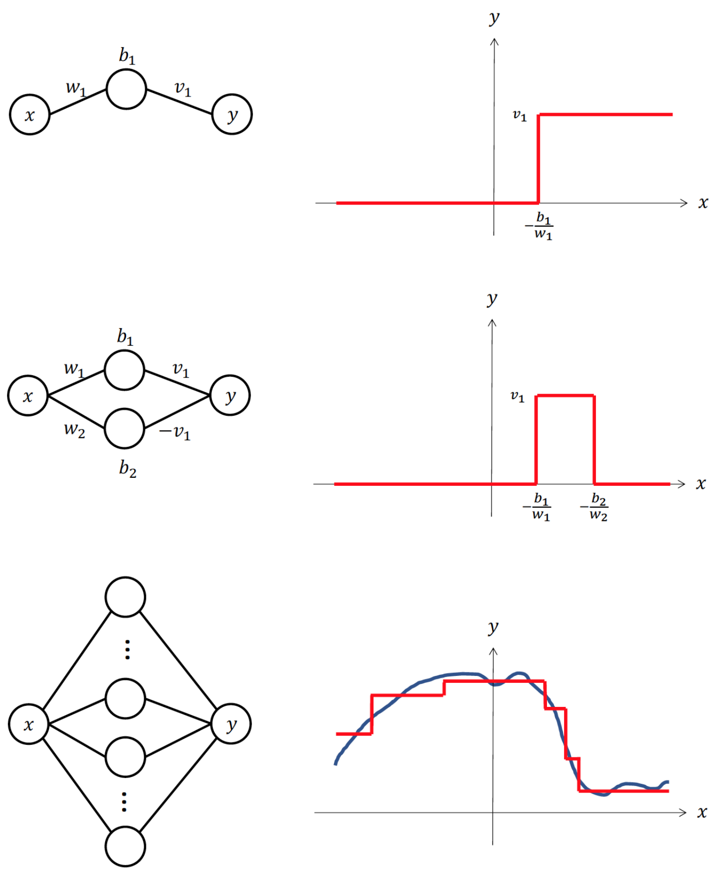
A perceptron is used in binary classification. It takes in some input $\boldsymbol{x}$, aggregates the weighted sum and returns 1 if the aggregated sum exceeds a threshold, and returns 0 otherwise.

$$f(\boldsymbol{w}^\top \boldsymbol{x} + b) = \begin{cases} 1, & \boldsymbol{w}^\top \boldsymbol{x} + b \geq 0 \\ 0, & \boldsymbol{w}^\top \boldsymbol{x} + b < 0 \end{cases}$$

1.2 Universal Approximation Theorem

The **Universal Approximation Theorem** states that a perceptron with one hidden layer of finite width can approximate any continuous function.

The intuition behind this is that one perceptron can create a step function. Combining two perceptrons aggregates the two step function to create a bump. By adding more and more perceptrons, it is possible to create many bumps to approximate the continuous function.



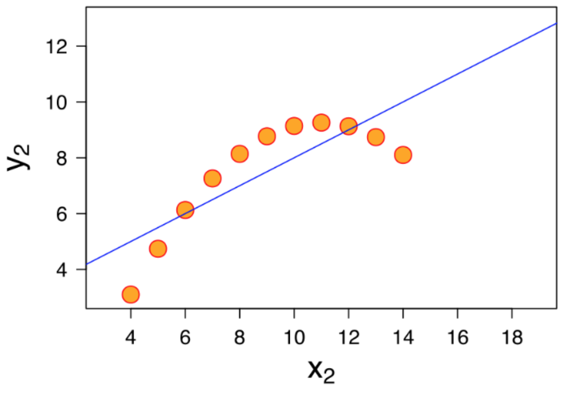
2 Generalisation Error

The three main components in deep learning are **model**, **data**, and **optimisation**. All of these components introduces error to the algorithm.

2.1 Approximation Error

Approximation error comes from the model. When forming predictive models, we usually restrict the set of models to some families. However, if the nature of the data generating mechanism does not follow these rules, even the best predictor in the family will not be able to predict the data.

For example, if we restrict ourselves to use only linear models. Even the best model will not be able to express the data.



Approximation Error can be reduced by enlarging the family of predictors. However, capturing more more complexities often leads to over-fitting and hence increases Estimation error.

2.2 Estimation Error

2.2.1 Generative and Discriminative Classifiers

There are two different algorithms to classify objects: Generative and Discriminative. A Generative Model ?explicitly models the actual distribution of each class. A Discriminative model ?models the decision boundary between the classes.

As an example, suppose your task is to classify different languages. You can achieve this either by:

1. Learning each of the languages and then classify the language using the knowledge (Generative)

2. Learning the differences between each linguistic model without actually learning the language (Discriminative)

Mathematically, Generative model learns the joint probability distribution $p(x, y)$, and then uses Bayes Theorem to find $p(y \mid x)$.

$$p(y \mid x) = \frac{p(x, y)}{p(x)}$$

On the other hand, Discriminative model learns the conditional probability distribution $p(y \mid x)$ directly.

2.2.2 Generalisation Gap

Estimation error (Generalisation Gap) is the difference between a model's performance on training data and its performance on unseen data drawn from the same distribution.

Ideally, a model should reduce the estimation error L during training. In reality, this model is updated using training error $\hat{L}$, which is only an estimate of L. This difference leads to estimation error.

One way to minimise estimation error is to increase the size of training data. According to **Hoeffding inequality**, using larger sample size n can give better estimation. In other words, a using more training data can better estimate $\hat{L} \approx L$ and hence reduce the generalisation gap.

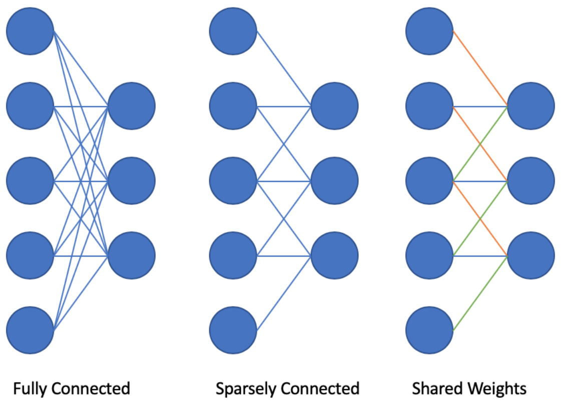
$$P\left(\mid L - \hat{L} \mid > \epsilon\right) \leq 2e^{-2\epsilon^2 n}$$

2.3 Optimisation Error

Optimisation error occurs when the optimiser fails to find the optimal solution, for example stuck at suboptimal solution. This can be reduced by running more iterations and increase training time.

3 Convolution Neural Network

A Convolution Neural Network can be used to differentiate different images. An image is just a matrix of pixel values, one approach is to flatten the matrix and feed it to a fully connected neural network.



In a fully connected neural network, all the nodes in adjacent layers are fully connected with each other. This becomes impractical when the image size is huge.

One way to reduce to number of parameters is to use a sparse connected neural network. The nodes in the second layer is only connected to parts of the first layer. This effectively reduces the number of connections hence the number of parameters.

To further reduce the number of parameters, each node in the next layer shares the same weight. This is the same as taking one node and applying convolution to the first layer.

3.1 Convolution Layer

The first layer in a CNN is always a Convolutional Layer. In the convolution layer, a filter is slide around the input image and is multiplying the values in the filter with the original pixel values of the image. These multiplications are all summed up to get a single number. A very important note is that the depth of this filter has to be the same as the depth of the input. The output produced by the filter is a feature map. We can apply multiple filters to the image to obtain multiple feature maps.

There are multiple parameters we can define for the convolution layer

- padding: adding zero padding to the borders of the input image
- stride: The number of pixels shifts over the input matrix
- kernel size: The width and height of the filters

Given an input of size (N,C,H,W). A convolution layer with padding p, stride s, kernel size (HH,WW) and F filters with produce output with shape (N,F,H',W'), where

$$H' = \frac{H - HH + 2p}{s} + 1 \qquad W' = \frac{W - WW + 2p}{s} + 1$$

3.2 Pooling Layer

Pooling layers provide an approach to down sampling feature maps by summarizing the presence of features in patches of the feature map. Two common pooling methods are average pooling and max pooling that summarize the average presence of a feature and the most activated presence of a feature respectively.

Given an input of size (N, C, H, W). A pooling layer with kernel size (HH, WW) with produce output with shape (N, C, H', W'), where

$$H' = \frac{H - HH}{HH} + 1 \qquad\qquad W' = \frac{W - WW}{WW} + 1$$

3.3 Shift Invariance and Shift Equivariance

A function f is invariant to a transformation T if it returns the same output even if the input data is transformed.

$$f(T(x)) = f(x)$$

A function f is equivariant to a transformation T if applying the transformation to the input is the same as applying the transformation to the output.

$$f(T(x)) = T(f(x))$$

The convolution operator is shift equivariance because convolving then shifting an image has the same result as shifting the image then applying convolution.

On the other hand, pooling layer is shift invariant.

4 Generative Models

A Generative Model is a powerful way of learning any kind of data distribution using unsupervised learning. All types of generative models aim at learning the true data distribution of the training set so as to generate new data points with some variations.

Two of the most commonly used and efficient approaches are Variational Autoencoders (VAE) and Generative Adversarial Networks (GAN). VAE aims at maximizing the lower bound of the data log-likelihood and GAN aims at achieving an equilibrium between Generator and Discriminator.

4.1 Variational Autoencoders (VAE)

A variational autoencoder consists of an encoder, a decoder, and a loss function.

The encoder $q_\theta(z \mid x)$ is a neural network with weights θ, it takes in a data point x and outputs a distribution in the latent space. We can sample from this distribution to get noisy values of the representations z.

The decoder $p_\phi(x \mid z)$ is a neural network with weights ϕ, it takes in the representation z and outputs the parameters to the probability distribution of the data.

The loss function of the variational autoencoder is the negative log-likelihood with a regularizer.

$$\sum_{i=1}^{N} -\mathbb{E}_{z \sim q_\theta(z \mid x_i)} \left[\log p_\phi(x_i \mid z)\right] + \mathbb{KL}\left[q_\theta(z \mid x_i) \parallel p(z)\right]$$

The first term is called the reconstruction term, this term encourages the decoder to learn to reconstruct the data. The second term is a regularizer, it is expressed as the Kullback-Leibler divergence between the encoder's distribution $q_\theta(z \mid x)$ and $p(z)$. This divergence measures how much information is lost (in units of nats) when using q to represent p.

In variation encoder, $p(z)$ is specified as a standard Normal distribution with mean zero and variance one, $p \sim \mathcal{N}(0, \boldsymbol{I})$. With this regularisation term, we prevent the model to encode data far apart in the latent space and encourage as much as possible returned distributions to "overlap", satisfying this way the expected continuity and completeness conditions.

4.1.1 A Probabilistic Perspective

In the probability model framework, a variational autoencoder contains a specific probability model of data x and latent variables z. To generate a data point x_i:

1. Sample latent variable $z_i \sim p(z)$
2. Sample data variable $x_i \sim p(x \mid z)$

Under this generative process, our aim is to maximize the probability of each data in X

$$p(X) = \int p(X, z)\, dz = \int p(X \mid z) p(z)\, dz$$

Unfortunately, this integral is intractable as it needs to be evaluated over all configurations of latent variables. Instead, VAE uses Bayes' rule to rewrite

$$p(X) = \frac{p(X \mid z) p(z)}{p(z \mid X)}$$

VAE then uses a method called variational inference to infer $p(z \mid X)$. This method first model $p(z \mid X)$ using simpler distribution $q(z \mid X)$ which is easy to find and we try to minimize the difference between $p(z \mid X)$ and $q(z \mid X)$ using KL-divergence metric approach.

$$\log p_\theta(X) = \mathbb{E}_{z \sim q_\phi(z \mid X)} [\log p_\theta(X)]$$

$$
\begin{aligned}
&= \mathbb{E}_z\left[\log\left(\frac{p_\theta(X \mid z)p_\theta(z)}{p_\theta(z \mid X)}\frac{q_\phi(z \mid X)}{q_\phi(z \mid X)}\right)\right] \\
&= \mathbb{E}_z\left[\log p_\theta(X \mid z)\right] - \mathbb{E}_z\left[\log\frac{q_\phi(z \mid X)}{p_\theta(z)}\right] + \mathbb{E}_z\left[\log\frac{q_\phi(z \mid X)}{p_\theta(z \mid X)}\right] \\
&= \mathbb{E}_z\left[\log p_\theta(X \mid z)\right] - \mathbb{KL}\left(q_\phi(z \mid X) \parallel p_\theta(z)\right) + \mathbb{KL}\left(q_\phi(z \mid X) \parallel p_\theta(z \mid X)\right)
\end{aligned}
$$

From Jensen's Inequality, $\mathbb{KL}\left(q_\phi(z \mid X) \parallel p_\theta(z \mid X)\right) \geq 0$. Therefore, maximizing $\log p_\theta(X)$ is the same as maximizing the lower bound

$$\mathbb{E}_z\left[\log p_\theta(X \mid z)\right] - \mathbb{KL}\left(q_\phi(z \mid X) \parallel p_\theta(z)\right)$$

4.2 Generative Adversarial Network (GAN)

GAN consists of two adversarial networks. A neural network called the generator, generates new data instances and the other, the discriminator evaluates them for authenticity. Basically the task of the Generator is to generate natural looking images and the task of the Discriminator is to decide whether the image comes from the original training data.

We define a prior on input noise variables $p(z)$ and then the generator maps this to data distribution using a complex differentiable function with parameters θ.

Discriminator which takes in input x and using another differentiable function with parameters ϕ outputs a single scalar value denoting the probability that x comes from the training data.

The objective function for GAN is therefore

$$\min_G \max_D \mathbb{E}_{x \sim p_{data}(x)}\left[\log D(x)\right] + \mathbb{E}_{z \sim p_z(z)}\left[\log\left(1 - D(G(z))\right)\right]$$

When you train the discriminator, hold the generator values constant; and when you train the generator, hold the discriminator constant. Each should train against a static adversary.

When training the discriminator

$$\max_\phi \ \mathbb{E}_{x \sim p_{data}(x)}\left[\log D_\phi(x)\right] + \mathbb{E}_{z \sim p_z(z)}\left[\log\left(1 - D_\phi(G_\theta(z))\right)\right]$$

When training the generator

$$\min_\theta \ \mathbb{E}_{z \sim p_z(z)}\left[\log\left(1 - D_\phi(G_\theta(z))\right)\right]$$

But there is a problem in optimizing generator objective. At the start of the game when the generator hasn?t learned anything, the gradient is usually very small and when it is doing very well, the gradients are very high. But we want the opposite behaviour, therefore we usually rewrite the second equation to

$$\max_\theta \ \mathbb{E}_{z \sim p_z(z)}\left[\log\left(D_\phi(G_\theta(z))\right)\right]$$

4.2.1 Deep Convolutional Generative Adversarial Networks (DCGAN)

This network takes as input 100 random numbers drawn from a uniform distribution and outputs an image of desired shape. The generator network uses many deconvolutional layers to map the input noise to the desired output image. The authors showed that the generators have interesting vector arithmetic properties using which we can manipulate images in the way we want.

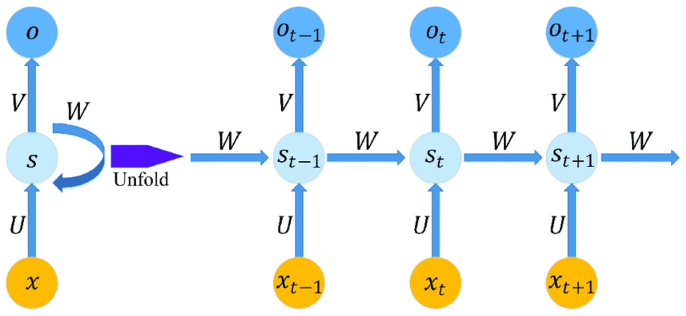
4.2.2 Conditional GAN

One of the most widely used variation of GANs is conditional GAN which is constructed by simply adding conditional vector along with the noise vector. Traditional GAN generates images randomly from random samples of noise z. Conditional GAN adds extra information, for example the class, to generate a specific type of image. By conditioning the model on additional information which is provided to both generator and discriminator, it is possible to direct the data generation process.

5 Recurrent Neural Networks

Traditional Neural Networks cannot keep track of previous information to make predictions. Recurrent Neural Networks address this issue by adding loops to the networks, allowing information to persist.

5.1 Vanilla RNN



At time t, the RNN takes in some input x_t and a state from from the previous time step s_{t-1} and outputs a new state s_t. The new state s_t is used as the input for the next time step.

In vanilla RNN, the new state is computed as

$$s_t = tanh(Ws_{t-1} + Ux_t)$$

At each state, the RNN would produce an output as well.

$$o_t = Vs_t$$

5.1.1 BackPropagation Through Time

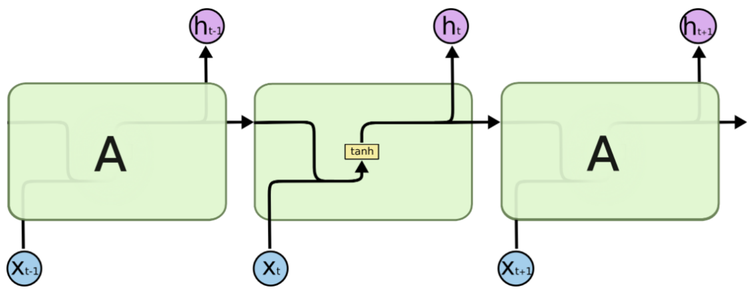
Backpropagation Through Time (BPTT) is the algorithm that is used to update the weights in the recurrent neural network. The goal is to calculate the gradients of the error with respect to our parameters U, V, and W. We have to sum up all the gradients at each time steps in one training.

$$\frac{\partial L}{\partial W} = \sum_{j=0}^{T} \sum_{k=0}^{j} \frac{\partial J_j}{\partial o_j} \frac{\partial o_j}{\partial s_j} \left(\prod_{t=k+1}^{j} \frac{\partial h_t}{\partial h_{t-1}} \right) \frac{\partial h_k}{\partial W}$$

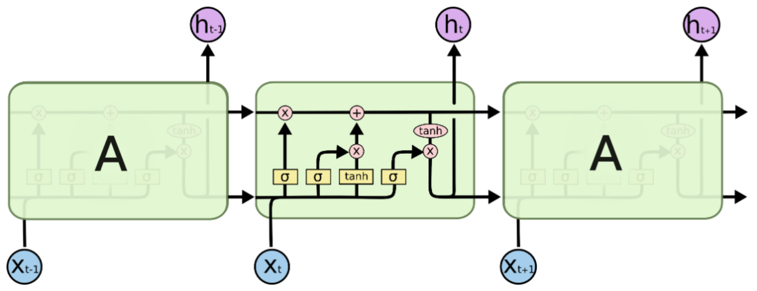
One problem with vanilla RNN is that repeated matrix multiplications during back propagation leads to vanishing and exploding gradients.

5.2 Long Short Term Memory Networks (LSTM)

LSTMs are special kind of RNN capable of learning long-term dependencies. In standard RNNs, the repeating module has a simple tanh layer.

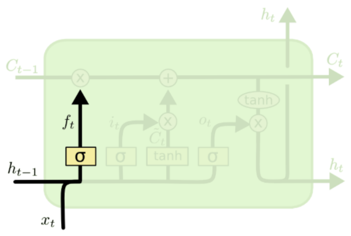


In LSTM, the repeating module has a different structure, it uses four neural networks that interacts with each other.



The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged. The LSTM does have the ability to remove or add information to the cell state using **gates**.

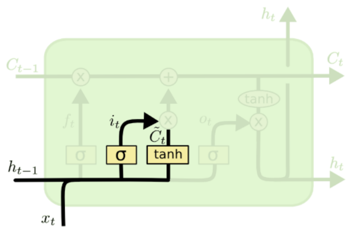
5.2.1 Forget Gate



The first step in our LSTM is to decide what information we're going to throw away from the cell state. This is decided by the **Forget Gate**. The gate outputs a number between 0 and 1 to indicate how much information to keep.

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

5.2.2 Input Gate



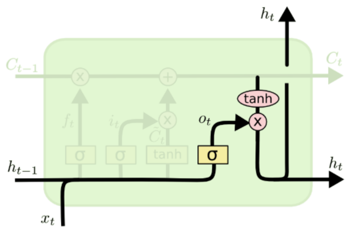
The next step is to decide what new information we're going to store in the cell state. A sigmoid layer called the **Input Gate** decides which values to update, a tanh layer creates a vector of new candidate values, $\tilde{C}_t$, that could be added to the state.

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

We can then combine the output of the forget gate to update state cell C_{t-1} to new state cell C_t.

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

5.2.3 Output Gate

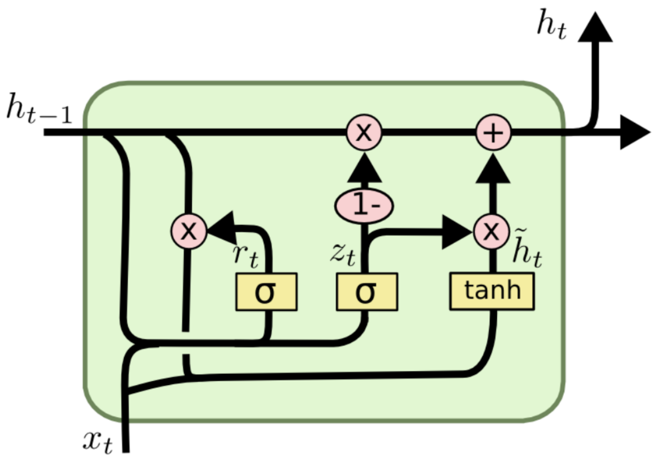


Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_0)$$
$$h_t = o_t \odot tanh(C_t)$$

5.3 Gated Recurrent Network

To solve the vanishing gradient problem of a standard RNN, GRU uses, so-called, update gate and reset gate. Basically, these are two vectors which decide what information should be passed to the output.



The first step is to calculate the **update gate**, z_t. The update gate helps the model to determine how much of the past information (from previous time steps) needs to be passed along to the future.

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$

The second step is to calculate the **reset gate**. This gate is used from the model to decide how much of the past information to forget.

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

Let's see how exactly the gates will affect the final output. We introduce a new memory content which will use the reset gate to store the relevant information from the past.

$$\tilde{h}_t = tanh(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h)$$

As the last step, the network needs to calculate the output using the input gate

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t$$