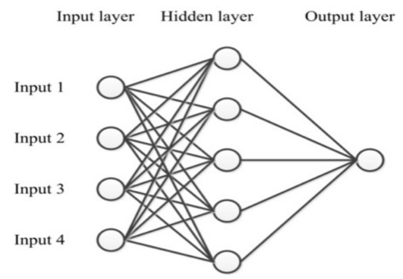
# Machine Learning Explained

*Lecture Notes of CS385, SJTU, taught by Prof. Quanshi Zhang*

Summarized and Written by Tony Fang (galaxies@sjtu.edu.cn, tony.fang.galaxies@gmail.com)

# Chapter 4. Neural Network Basics

## 4.1 Two-Layer Perceptron and Rectified Linear Unit Activation



A perceptron seeks to separate the positive examples and the negative examples by projecting them onto a vector $\beta$, *i.e.*, separating them using a hyperplane that is perpendicular to $\beta$. If the data are not linearly separable, a perceptron won't work. We may need to transform the original data into some feature dimensions so that the features can be linearly separated. One way to solve the problem is to generalize the perceptron into multi-layer perceptron. This structure is also called feedforward neural network.

This neural network is logistic regression on top logistic regressions. $y_i \in \{0,1\}$ follows a logistic regression on $h_i = (h_{ik},\ k = 1,2,\cdots,d)^T$, and each $h_{ik}$ follows a logistic regression on $X_i = \left(x_{ij},\ j = 1,2,\cdots,p\right)^T$, that is,

$$y_i \sim B(p_i)$$

$$p_i = \mathrm{sigmoid}(h_i^T\beta) = \mathrm{sigmoid}\left(\sum_{k=1}^{d} \beta_k h_{ik}\right)$$

$$h_{ik} = \mathrm{sigmoid}(X_i^T\alpha_k) = \mathrm{sigmoid}\left(\sum_{j=1}^{p} \alpha_{kj} x_{ij}\right)$$

**Back-propagation[1].**

First, let us derive the log-likelihood for the two-layer perceptron.

$$\log \mathfrak{L}(\alpha,\beta) = \sum_{i=1}^{n}[y_i \log p_i + (1-y_i)\log(1-p_i)] = \sum_{i=1}^{n}[y_i A_i - \log(1+\exp A_i)]$$

where $A_i = \sum_{k=1}^{d}\beta_k h_{ik} = \beta^T h_i$. Notice we use $\mathfrak{L}(\alpha,\beta)$ to represent the log-likelihood $\log \mathfrak{L}(\alpha,\beta)$ in the following derivations for convenience. Hence,

$$\mathfrak{L}(\alpha,\beta) = \sum_{i=1}^{n}\left[y_i \sum_{k=1}^{d}\beta_k h_{ik} - \log\left(1+\exp\sum_{k=1}^{d}\beta_k h_{ik}\right)\right]$$

whose gradients are

$$\frac{\partial \mathfrak{L}}{\partial \beta} = \sum_{i=1}^{n}\left(y_i h_i - \frac{\exp A_i}{1+\exp A_i} h_i\right) = \sum_{i=1}^{n}(y_i - p_i) h_i$$

and

$$\frac{\partial \mathfrak{L}}{\partial \alpha_k} = \frac{\partial \mathfrak{L}}{\partial h_{\cdot,k}} \cdot \frac{\partial h_{\cdot,k}}{\partial \alpha_k} = \sum_{i=1}^{n}\frac{\partial \mathfrak{L}}{\partial h_{ik}} \cdot \frac{\partial h_{ik}}{\partial \alpha_k} = \sum_{i=1}^{n}(y_i - p_i)\beta_k \cdot h_{ik}(1-h_{ik})X_i$$

Notice that we only use the chain rule here to derive the gradients. Again, the gradient descent algorithm learns from error term $(y_i - p_i)$. The chain rule back-propagates the error to assign the blame on $\beta$ and $\alpha$ in order for those parameters to update. If the current network makes a mistake on the $i$-th sample, $\beta$ will change to be more aligned with $h_i$, where each $\alpha_k$ also changes to be more aligned with $X_i$. The amount of change depends on $\beta_k$ as well as $h_{ik}(1-h_{ik})$, which measures how big a role played by $X_i^T\alpha_k$ in predicting $y_i$. If it plays a big role, then $\alpha_k$ should receive the blame and make change.

[1] Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." nature 323.6088 (1986): 533-536.

**Rectified Linear Unit (ReLU)**[2, 3].

In modern neural network, the non-linearity is often provided by the rectified linear unit, *a.k.a.* ReLU, which has the form of $f(a) = \max(0, a)$. We can use ReLU activation function to substitute Sigmoid activation function of the first layer. Therefore,

$$y_i \sim B(p_i)$$

$$p_i = \text{sigmoid}(h_i^T \beta) = \text{sigmoid}\left(\sum_{k=1}^{d} \beta_k h_{ik}\right)$$

$$h_{ik} = \text{ReLU}(X_i^T \alpha_k) = \max\left(0, \sum_{j=1}^{p} \alpha_{kj} x_{ij}\right)$$

For ReLU activation function, we should replace the gradients of Sigmoid function $h_{ik}(1 - h_{ik})$ by the binary predicate function $[h_{ik} > 0]$, which has value 1 for the true predicate and 0 otherwise.

$$\frac{\partial \mathfrak{L}}{\partial \alpha_k} = \sum_{i=1}^{n} (y_i - p_i)\beta_k \cdot [h_{ik} > 0] X_i$$

We can dive deeper into the ReLU activation function. Let us compare it with the Sigmoid activation function we used before. The Sigmoid activation function saturates at the two ends, where the derivatives are close to 0. This can cause the vanishing gradient problem in back-propagation, which makes the network unable to learn from errors. The ReLU activation function does not saturate for big positive input, which indicates the existence of a certain patten. This helps avoid the vanishing gradient problem.

If we focus on the spline regression introduced in Chapter 3.4, we can find it has ReLU activation functions in its expression. Therefore, the neural network with ReLU can be viewed as a high-dimensional spline, or piecewise linear mapping. If there are many layers in the neural network, the number of linear pieces is exponential in the number of layers. It can approximate highly non-linear mapping by patching up the large number of linear pieces.

## 4.2 Multi-layer Neural Network

The multi-layer perceptron or the feedforward neural network consists of an input layer, an output layer, and multiple hidden layers in between. Each layer can be represented by a vector, which is obtained by multiplying the layer below by a weight matrix, plus a bias vector, and then transforming each element of the resulting vector by activation functions like Sigmoid, ReLU, *etc*. More formally, the network has the following structure:

$$h_l = f_l(s_l), \qquad s_l = W_l h_{l-1} + b_l$$

for $l = 1, 2, \dots, L$, where $l$ denotes the number of the layer, with $h_0 = X$ being the input vector, and $h_L$ is used to predict $Y$, which is the output label corresponding to $X^T\beta$ in the previous linear model chapter. $W_l$ and $b_l$ are the weight matrix and the bias vector respectively, and $f_l$ is the element-wise transformation activation function such as Sigmoid and ReLU, *i.e.*, $h_{lk} = f_l(s_{lk})$ for all $k$.

**Chain Rule of Matrix Form**.

Suppose $Y = (y_i)_{m\times 1}$, $X = (x_j)_{n\times 1}$, and $Y = h(X)$. We can define

$$\frac{\partial Y}{\partial X^T} = \left(\frac{\partial y_i}{\partial x_j}\right)_{m\times n}$$

To understand the definition, we can treat $\partial/\partial X^T$ as the operator matrix defined as

$$\frac{\partial}{\partial X^T} = \left(\frac{\partial}{\partial x_j}\right)_{1\times n}$$

Therefore, the definition can be treated as

$$\frac{\partial Y}{\partial X^T} = Y \times \frac{\partial}{\partial X^T} = (y_i)_{m\times 1} \times \left(\frac{\partial}{\partial x_j}\right)_{1\times n} = \left(\frac{\partial y_i}{\partial x_j}\right)_{m\times n}$$

[2] Nair, Vinod, and Geoffrey E. Hinton. "Rectified linear units improve restricted boltzmann machines." *ICML*. 2010.

[3] Maas, Andrew L., Awni Y. Hannun, and Andrew Y. Ng. "Rectifier nonlinearities improve neural network acoustic models." *Proc. ICML*. Vol. 30. No. 1. 2013.

Naturally, the transpose of the previous operator matrix $\partial/\partial X$ is defined as

$$\frac{\partial}{\partial X} = \left(\frac{\partial}{\partial x_j}\right)_{n\times 1}$$

Then, only for scalar $Y$, we can define $\partial Y/\partial X$ as an $n \times 1$ vector defined as

$$\frac{\partial Y}{\partial X} = \frac{\partial}{\partial X} \times Y = \left(\frac{\partial}{\partial x_j}\right)_{n\times 1} \times Y = \left(\frac{\partial Y}{\partial x_j}\right)_{n\times 1}$$

There are some multiplication rules of the partial notation, listed as follows.

1. If $Y = AX$, then $y_i = \sum_k a_{ik}x_k$, hence, $\partial Y/\partial X^T = A$;
2. If $Y = X^T SX$, where $S$ is symmetric, then $\partial Y/\partial X^T = 2SX$;
3. (Special case of case 2) If $S = I$, which indicates that $Y = \|X\|^2$, then $\partial Y/\partial X^T = 2X$.

The chain rule of the matrix form is as follows. If $Y = h(X)$ and $X = g(Z)$, then

$$\frac{\partial y_i}{\partial z_j} = \sum_k \frac{\partial y_i}{\partial x_k} \cdot \frac{\partial x_k}{\partial z_j}$$

which indicates that

$$\frac{\partial Y}{\partial Z^T} = \frac{\partial Y}{\partial X^T}\frac{\partial X}{\partial Z^T}$$

**Multi-layer Back-propagation**.

For multi-layer network, the chain rule means that

$$\frac{\partial \mathfrak{L}}{\partial s_l^T} = \frac{\partial \mathfrak{L}}{\partial h_l^T}\frac{\partial h_l}{\partial s_l^T} = \frac{\partial \mathfrak{L}}{\partial h_l^T} f_l'$$

$$\frac{\partial \mathfrak{L}}{\partial h_{l-1}^T} = \frac{\partial \mathfrak{L}}{\partial h_l^T}\frac{\partial h_l}{\partial s_l^T}\frac{\partial s_l}{\partial h_{l-1}^T} = \frac{\partial \mathfrak{L}}{\partial h_l^T} f_l' W_l$$

Where $f_l'$ is a diagonal matrix because $f_l$ is elementwise, which means the $k$-th diagonal element of $f_l'$ is $\partial h_{lk}/\partial s_{lk}$, *i.e.*, $h_{lk}(1 - h_{lk})$ for Sigmoid activation function and $[h_{lk} > 0]$ for ReLU activation function.

Notice that $\mathfrak{L}$ is a scalar, then we can transpose the above equation and have equation (*):

$$\frac{\partial \mathfrak{L}}{\partial h_{l-1}} = W_l^T f_l' \frac{\partial \mathfrak{L}}{\partial h_l}$$

Let $W_{lk}$ be the $k$-th row of $W_l$, and let $s_{lk}$ be the $k$-th element of $s_l$, then

$$\left(\frac{\partial \mathfrak{L}}{\partial W_{lk}}\right)_{1\times K} = \left(\frac{\partial \mathfrak{L}}{\partial s_{lk}}\right)_{1\times 1} \times \left(\frac{\partial s_{lk}}{\partial W_{lk}}\right)_{1\times K} = \left(\frac{\partial \mathfrak{L}}{\partial s_{lk}}\right)_{1\times 1} \times (h_{l-1}^T)_{1\times K}$$

Combining all rows, then we can have equation (**) as follows.

$$\left(\frac{\partial \mathfrak{L}}{\partial W_l}\right)_{K\times K} = \left(\frac{\partial \mathfrak{L}}{\partial s_l}\right)_{K\times 1} \times (h_{l-1}^T)_{1\times K} = (f_l')_{K\times K} \times \left(\frac{\partial \mathfrak{L}}{\partial h_l}\right)_{K\times 1} \times (h_{l-1}^T)_{1\times K}$$

Define

$$\Delta h_l = \frac{\partial \mathfrak{L}}{\partial h_l}, \qquad \Delta W_l = \frac{\partial \mathfrak{L}}{\partial W_l}, \qquad D_l = f_l'$$

and then according to (*) and (**) we have

$$\Delta h_{l-1} = W_l^T D_l \Delta h_l$$
$$\Delta W_l = D_l \Delta h_l h_{l-1}^T$$

Similarly, we can also include $b_l$ by adding an constant intercept term 1 to $h_{l-1}$, which means

$$(\Delta W_l, \Delta b_l) = \Delta(W_l, b_l) = D_l \Delta h_l (h_{l-1}^T, 1)$$

and then

$$\frac{\partial \mathfrak{L}}{\partial b_l} = \Delta b_l = D_l \Delta h_l$$

Then, we can learn $W_l$ and $b_l$ by gradient descent, which is again learning from mistakes by error back-propagation. And the back-propagation process is illustrated as follows.

$$\Delta h_L \to \Delta h_{l-1} \to \cdots \to \Delta h_1, \qquad \Delta h_l \to \Delta W_l, \Delta b_l$$

which explains that back-propagation takes twice time comparing to the forwarding process of neural network.

## 4.3 Convolutional Neural Network Basics: Convolution, Pooling, Fully Connected Layer

In the neural network, $h_l = f_l(W_l h_{l-1} + b_l)$, the linear transformation $s_l = W_l h_{l-1} + b_l$ that maps $h_{l-1}$ to $s_l$ can be highly structured. One important structure is the convolutional neural network[4, 5], which is also known as CNN or ConvNet, where $W_l$ and $b_l$ have a convolutional structure.

**Convolution**.

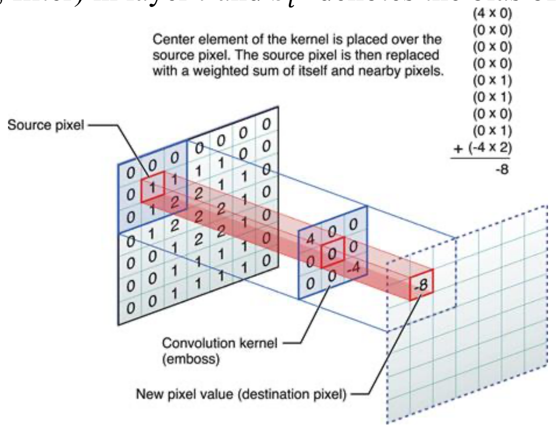
Specifically, a convolutional layer $h_l$ is organized into a number of feature maps. Each feature map is also called a channel and is obtained by a filter or a kernel operating on $h_{l-1}$, which is also organized into a number of feature maps. Each filter is a local weighted summation plus a bias, followed by a non-linear activation transformation, that is,

$$s_l = W_l \otimes h_{l-1} + b_l, \qquad h_l = f(s_l)$$

in which $\otimes$ is the notation of convolution, which means that

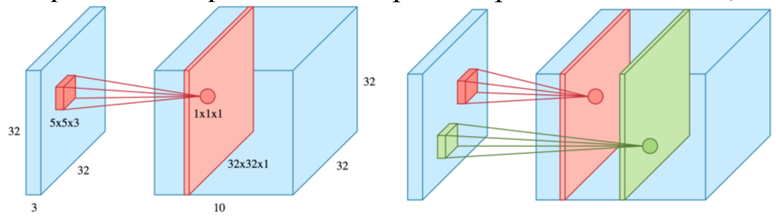
$$s_{l,u,v,w} = b_l^w + \sum_{i=1}^{R}\sum_{j=1}^{R}\sum_{k=1}^{C} W_{l,i,j,k}^w h_{l-1,t(u-1)-p+i,t(v-1)-p+j,k}$$

where kernel size is $R \times R \times C$ and $t$ denotes the stride, $p$ denotes the padding, $W_{l,i,j,k}^w$ denotes the unit $(i,j,k)$ of the $w$-th kernel (output channel, filter) in layer $l$ and $b_l^w$ denotes the bias of the $w$-th kernel in layer $l$.


Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

The above figure shows the convolution operation – the local weight summation. We convolve an input feature map with a $3 \times 3$ filter, to obtain an output feature map. The value of each pixel of the output feature map is obtained by a weighted summation of pixel value of the $3 \times 3$ patch of the input feature map around this pixel. We apply the weighted summation around each pixel with the same weights to obtain the feature map.

If there are multiple input feature maps, *i.e.*, multiple input channels, the weighted summation is also over the multiple feature maps. We can apply different filters to the same set of input feature maps, and then we can get different output feature maps. Each output feature map corresponds to one filter, as illustrated below.



After the convolution operation (the weighted summation), we may also add a bias and apply a non-linear transformation to make the filter non-linear, such as Sigmoid activation function or ReLU activation function.

**Pooling**.

After obtaining the feature maps in $h_l$, we may perform max-pooling, *i.e.*, for each feature map, at each pixel, we replace the value of this pixel by the maximum of the $k \times k$ patch around this pixel, where $k$ is the kernel size of the pooling layer. We may also apply average-pooling or min-pooling, which simply replace the value of the pixel by the average value or minimum of the $k \times k$ patch around this pixel. We call max-pooling, min-pooling and average-pooling as pooling in general.

[4] Fukushima, Kunihiko, and Sei Miyake. "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition." *Competition and cooperation in neural nets*. Springer, Berlin, Heidelberg, 1982. 267-285.
[5] LeCun, Yann, et al. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE* 86.11 (1998): 2278-2324.

**Sub-sampling**.

The mapping from $h_{l-1}$ to $h_l$ may involves sub-sampling to reduce the size of the feature maps. The sub-sampling process can be found in convolutional layer, pooling layer, or directly as a sub-sampling layer. The stride in convolutional layer and pooling layer indicates the sub-sampling ratio. For example, after obtaining the feature map by a filter, we can partition the feature map into $t \times t$ blocks where $t$ is the stride we introduced before. Within each block, we only keep the upper left pixel (or the middle pixel if you like). This will reduce the width and height of the feature map by half. $h_l$ may have more channels than $h_{l-1}$ since each pixel covers a bigger spatial extent than each element of $h_{l-1}$, and there are more patterns of bigger spatial extent.

**Fully Connected Layer**.

The output feature map may also be $1 \times 1$, whose value is a weighted sum of all elements in $h_{l-1}$. Then, $h_l$ may consists of a number of $1 \times 1$ feature maps. It is called fully connected layer because each element of $h_l$ is connected to all elements in $h_{l-1}$. Hence, fully connected layers are special convolutional layers, and we can also say that convolutional layer is a special kind of the general layer $s_l = W_l h_{l-1} + b_l$ we discussed before.

A convolutional neural network consists of multiple layers of convolutional and fully connected layers, with max-pooling and sub-sampling between the layers. The network can be learnt by back-propagation, as we have discussed in Chapter 4.2.

## 4.4 Convolutional Neural Network: Soft-max Layer, Batch Normalization, Up-convolution

**Soft-max Layer**.

Suppose we want to classify the input $X_i$ into one of $K$ categories. We can build a network whose top layer is a $K$ dimensional vector $h_i = (h_{ik})_{k=1,2,\cdots,K}$. Then we output the probability that the input $X_i$ belongs to the category $k$ as

$$p_{ik} = \frac{e^{h_{ik}}}{\sum_{k'=1}^{K} e^{h_{ik'}}}$$

which is called a soft-max layer. Actually, we have discussed it in Chapter 1.10. In testing or inference stage, we can simply classify $X_i$ to category $k$ whose $h_{ik}$ is the maximum in $h_i$, which is called a hard-max.

As we discussed in Chapter 1.10, for discriminative classification model, the soft-max layer enables the model to learn from errors. Then, we can back-propagate this error using the chain rule of the previous subsection to obtain $(\Delta W_l, \Delta b_l)$ so that the network can be updated.

The maximum likelihood estimation with a soft-max layer is related to the least square regression between $Y$ and $p$. For regression, the least square loss is $L(y_i, h_i) = \|y_i - h_i\|^2$, and the derivative with respect to $h_i$ is $-2(y_i - h_i) = -2e_i$, where $e_i$ is the error term.

**Batch Normalization**[6].

When training the neural network by back-propagation, the distribution of $h_l$ keeps changing because the parameters keep changing. This may cause problem in training. We can stabilize the distribution by batch normalization. That is, between $h_{l-1}$ and $h_l$, we can add a batch normalization layer. For simplicity, let $x$ be the input to the batch normalization layer, and let $y$ be the output of the batch normalization layer. For instsance, $x = h_{l-1}$ and $y$ becomes the normalized version of $h_{l-1}$ to be fed into the layer for computing $h_l$.

With slight abuse of notation, let $x_i$ be a sample element of $h_{l-1}$ which consists of $d$ dimensions. Assume we have $n$ training samples, so that we have $(x_i)_{i=1,2,\cdots,n}$. The batch normalization layer is defined as follows.

$$\mu_d = \frac{1}{n}\sum_{i=1}^{n} x_{id}, \qquad \sigma_d^2 = \frac{1}{n}\sum_{i=1}^{n}(x_{id} - \mu_d)^2$$

$$y_{id} = \beta_d + \gamma_d \hat{x}_{id} = \beta_d + \gamma_d \frac{x_{id} - \mu_d}{\sigma_d}$$

Therefore, we can stabilize the distribution of each element of $h_{l-1}$ during the training process. Note that in batch normalization layer, $\beta_d$ and $\gamma_d$ are parameters to be learnt. More importantly, $\mu$ and $\sigma^2$ are functions of the whole batch. When we do chain rule calculations for back-propagation, we need to compute the derivatives of $\mu$ and $\sigma^2$ with respect to all the $x_i$. It is as if the whole batch becomes a single training example.

[6] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." *International conference on machine learning*. PMLR, 2015.

When we add a batch normalization layer to a convolutional neural network, we need to learn $\beta, \gamma$ and compute $\mu, \sigma^2$ for each channel of $x_i$, rather than for each neural activation in $x_i$. Let $x_i$ be a $m \times m \times d$ tensor, then we need to compute $d$ sets of $\mu, \sigma^2$ for $d$ channels. In particular, we need to average over $nm^2$ numbers to compute $\mu$ and $\sigma^2$.

Sometimes, we can apply momentum approach in the batch normalization layers. If we only focus on the current batch, there can be huge difference of values among the batches. Therefore, we add momentum to keep some parts of the original value and add the current value to calculate $\mu, \sigma^2$ in a more precise way.

**Up-sampling**.

On the contrary to sub-sampling, up-sampling increase the size of the feature maps. A simple up-sampling method is zero-padding, *i.e.*, fill 0 in the pixels that is not corresponds to the previous feature maps. There are some other methods that requires interpolation, such as bilinear interpolation, nearest neighbor interpolation.

**Up-convolution**.

Up-convolution is also known as deconvolution and convolution transpose in some famous framework. It consists of two parts: an up-sampling layer and a convolutional layer. The up-sampling layer increases the size of the feature maps, while the convolutional layer mixes the pixels with its adjacent pixels to fill in the 0s of the up-sampled feature maps if we use zero-padding in up-sampling.

Notice that if we use zero-padding in the up-sampling stage, then we had better to use kernels of even size in the following convolution stage, such as a $2 \times 2$ kernel and a $4 \times 4$ kernel. This is because if we use kernels of odd size like $3 \times 3$, then $3 \times 3$ area can have 4, 2, or 1 non-zero padding value(s), depending on where you put the kernel in the feature map. For even-sized kernels, every area in the feature map has the same number of non-zero padding values, which makes the convolution reasonable and practical.

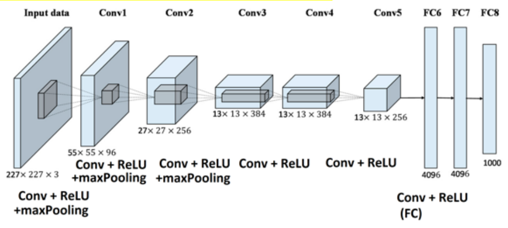
## 4.5 Convolutional Neural Network: General Designing Guidelines

This section introduces some special design guidelines of a convolutional neural network. Many of them are just experiences of the CNN designers. For CNN learners, we had better to follow the rules.

1. **Convolutional layer + convolutional layer is useless.** Since convolution is a linear transformation. Two convolutional layers are actually another linear transformation, which can be represented by only one convolutional layer. Therefore, consecutive convolutional layers are useless. The more reasonable design is convolutional layer + non-linear activation function such as ReLU + convolutional layer.
2. **Convolutional layer + batch normalization layer + convolutional layer is also useless**. Since batch normalization is also a linear transformation, we should add non-linear activation layers between them.
3. **ReLU activation layer + Soft-max layer makes no sense**. The ReLU activation layer makes the result value non-negative, while soft-max needs to have negative value, or the minimum value of $e^{h_{lk}}$ is actually 1, which makes no sense.
4. **Convolutional layer usually keeps the size unchanged, except for sub-sampling, *i.e.*, padding is necessary**. It's easy to understand, since changing the feature maps usually makes no sense in network.
5. **Up-convolutional layer usually has even-sized kernels**. The reason have been explained before.
6. **The number of channels usually have the form of $\mathbf{2^k}$**, which is a common habit.
7. **The number of batch size usually have the form of $\mathbf{2^k}$**, which is also a common habit.
8. **DO NOT use special numbers in hyper-parameters such as learning rate**. For example, if we set learning rate to 0.038247 and get the best performance, while a slight change of learning rate makes the performance drop quickly, then this model is not a good model since it is not robust. In practice, we usually set the learning rate to some ordinary numbers such as $10^{-3}$, $5 \times 10^{-2}$, *etc*.

## 4.6 Convolutional Neural Network Examples: AlexNet, VGG Net, Inception Net, ResNet

**AlexNet**[7].

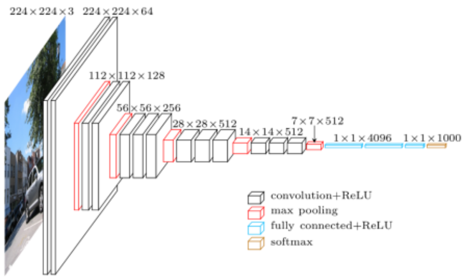


The AlexNet is the neural network that achieved the initial break-through on object recognition for the ImageNet dataset. It goes through several convolutional layers, followed by fully connected layers. It has 5 convolutional layers and 3 fully connected layers, plus a soft-max output layer, which can be seen in the right picture.

[7] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems* 25 (2012): 1097-1105.

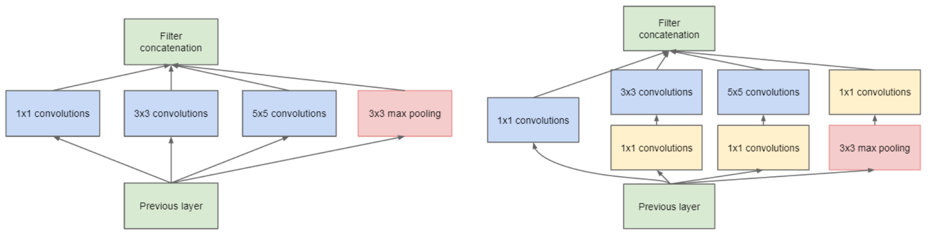
**VGG Net**[8].

The VGG Net is an improvement on the AlexNet. There are two versions, namely VGG-16 and VGG-19, which consists of 16 hidden layers and 19 hidden layers respectively. The VGG-19 has 144 million parameters. The filters of the VGG nets are all $3 \times 3$. Like AlexNet, it has three fully connected layers, plus a soft-max output layer. The architecture of VGG-19 is illustrated on the right.



**Inception Net**[9, 10].

The Inception Net took its name from the famous movie "inception" directed by Christopher Nolan, which has a line "we need to go deeper." The network makes extensive use of $1 \times 1$ filters, *i.e.*, for each feature map in $h_l$, each pixel value is a weighted summation of the pixel values of all the feature maps in $h_{l-1}$ at the same pixel, plus a bias and a non-linear transformation. The $1 \times 1$ filters serve to fuse the channels in $h_{l-1}$ at each pixel. The feature maps at each layer of the Inception Net are obtained by filters of sizes $1 \times 1, 3 \times 3$ and $5 \times 5$, as well as max-pooling. Then, the revised vision of inception module put $1 \times 1$ convolutions into the rest three branches to perform dimension reductions, as illustrated below.



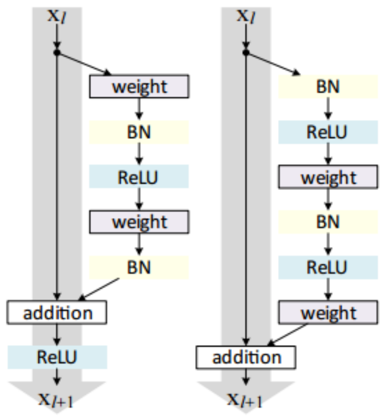
(a) Inception module, naïve version (b) Inception module with dimension reductions

**ResNet**[11].

A residual block in the ResNet is as follows. Let $x_l$ be the input to the residual block. Let $F(x_l)$ be the transformation of $x_l$ that consists of two rounds of weighted summation, batch normalization, and ReLU. We let

$$x_{l+1} = x_l + F(x_l)$$

as illustrated in the right figure. Notice that the left sub-figure is the original version of a residual block, and the right sub-figure is the revised version of a residual block, which has the form we wrote earlier. Actually, $F(x_l)$ models the residual of the mapping from $x_l$ to $x_{l+1}$, on top of the identity mapping. The following are some rationales for such a residual block.



(a) original (b) proposed

1. If we model $x_{l+1} = F(x_l)$, mathematically $F(x_l)$ parametrized by a set of weights may be the same as $x_l + F(x_l)$ parametrized by a different set of weights, computationally it can be difficult for gradient descent to learn the former than the latter. It can be such easier for stochastic gradient descent to find a good $F(x_l)$ in the residual form than in the original form.
2. The residual mapping models an iteration of an iterative algorithm or dynamic process, where $l$ denotes the time step $t$ instead of the real layer $l$, then it models the iterative refinement of the same layer over time.
3. With multiple residual blocks, we implicitly have an ensemble of networks. For instance, consider a sinple network $y = w_2 w_1 x$ where all the symbols are scalars. If we adopt a residual form $y = (1 + w_2)(1 + w_1)x$, we can expand it as $y = x + w_1 x + w_2 x + w_2 w_1 x$. Thus the residual form is an ensemble of 4 networks. We may also think of the residual form as an expansion like Taylor expansion.

[8] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556* (2014).

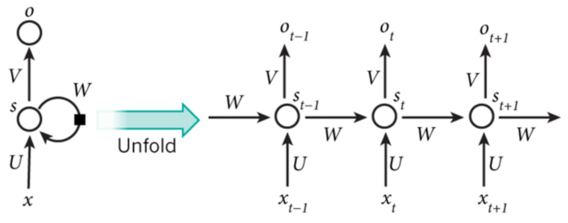
[9] Szegedy, Christian, et al. "Going deeper with convolutions." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015.

[10] Szegedy, Christian, et al. "Rethinking the inception architecture for computer vision." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.

[11] He, Kaiming, et al. "Deep residual learning for image recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.

## 4.7 Recurrent Neural Network

The vanilla version of recurrent neural network[12], which is also known as RNN, is illustrated as follows.



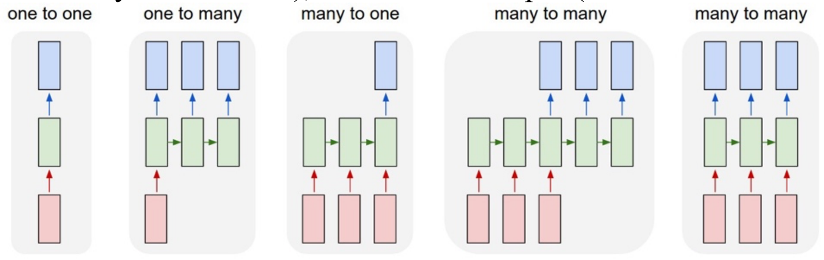
*A recurrent neural network and the unfolding in time of the computation involved in its forward computation. Source: Nature*

Formally, it can be written as

$$s_t = f(Ux_t + Ws_{t-1}), \qquad o_t = g(Vs_t)$$

where we use $U, V, W$ to denote weight matrices, and $f, g$ represent element-wise non-linear transformations.

There are many variations of inputs and outputs in RNN, as illustrated below. We can ignore the unnecessary output (such as many-to-one form), and feed zero input (such as one-to-many form).

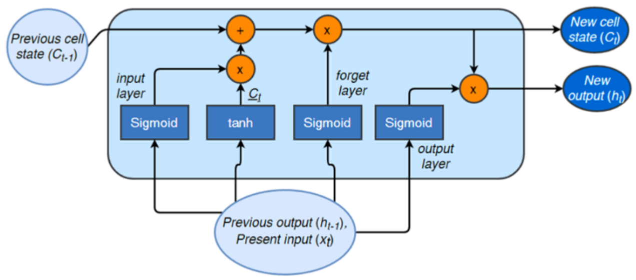


The training of RNN is again based on back-propagation we have discussed before, except that we need to back-propagate over time, but there is often vanishing (or exploding) gradient problem in RNN training.

## 4.8 Long-short Term Memory, Gated Recurrent Unit

**Long-short Term Memory[13].**

To solve the problems during RNN training, the Long-short Term Memory, also known as LSTM, is designed to overcome it by introducing memory cells. The architecture of LSTM is illustrated as follows.



Formally, it can be written as

1. $\Delta c_t = f\big(W_c(h_{t-1}, x_t)\big)$, which is the representation of input information $x_t$ combining old vector $h_{t-1}$.
2. $i_t = f\big(W_i(h_{t-1}, x_t)\big)$, which is a scalar input gate ranging in [0,1].
3. $f_t = f\left(W_f(h_{t-1}, x_t)\right)$, which is a scalar forget gate ranging in [0,1].
4. $c_t = f_t c_{t-1} + i_t \Delta c_t$, which is the cell state vector updating by the old cell state vector combining the forget gate, and the new input information combining the input gate.
5. $o_t = f\big(W_o(h_{t-1}, x_t)\big)$, which is a scalar output gate ranging in [0,1].
6. $h_t = o_t f(c_t)$, which is the hidden state vector updating by the cell state vector and the output gate.
7. $y_t = f(Wh_t)$ is the output of the current input.

The gates are like if-then statements in a computer program originally. They are made continuous by sigmoid activation function $f$ so that they are differentiable. If we regard $f_t = 1, i_t = 1$, ResNet can be regarded as a special case of LSTM.

[12] LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." *nature* 521.7553 (2015): 436-444.

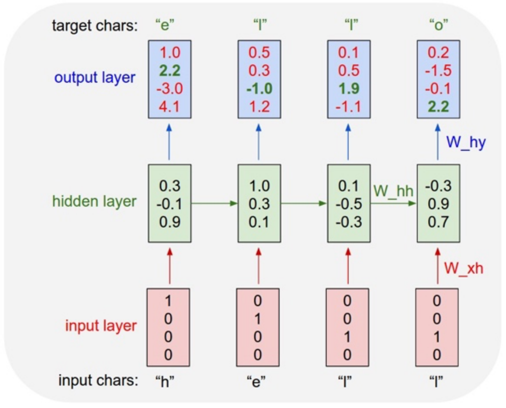
[13] Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." *Neural computation* 9.8 (1997): 1735-1780.

**Gated Recurrent Unit**[14].

The Gated Recurrent Unit, also known as GRU, is a simplification of LSTM by merging $c_t$ and $h_t$.

1. $z_t = f(W_z(h_{t-1}, x_t))$, which is a scalar update gate ranging in [0,1].
2. $r_t = f(W_r(h_{t-1}, x_t))$, which is a scalar reset gate ranging in [0,1].
3. $\tilde{h}_t = f(W(x_t, r_t h_{t-1}))$, which is the new information from the input.
4. $h_t = (1 - z_t)h_{t-1} + z_t \tilde{h}_t$, which is the hidden state.
5. $y_t = f(W_h h_t)$ is the output of the current input.

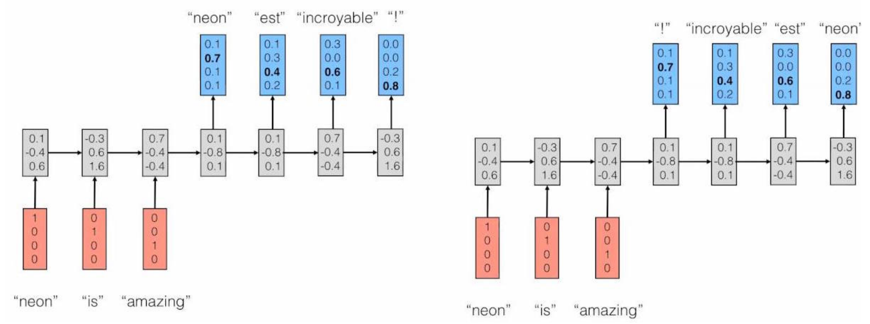
## 4.9 Encoder-Decoder Architecture and its Applications



The right figure shows an RNN for sequence generation, where $x_t$ is the current letter, and $y_t$ is the next letter. Each letter can be represented by a one-hot vector. The RNN maps the one-hot vector into a hidden vector, which is used to generate the next letter by soft-max probabilities. We can think of $x_t \to h_t$ as the encoder, and $h_t \to y_t$ as the decoder[15]. Because $h_t$ also depends on $h_{t-1}$, the $h_t$ is a thought vector that encodes all the relevant information of the past, that is, we have the following scheme:

$$\text{input} \to \text{thought vector} \to \text{output}$$

**Application: Machine Translation**.



The figure above shows the RNN for machine translation[16]. This time, each word is represented by a one-hot vector. We encode the input sentence into a thought vector by an encoding RNN. The thought vector is fed into a decoding RNN to generate the output sentence.

**Application: Image Captioning**[17].

Image captioning can be considered as a special case of machine translation, where the encoding RNN is replaced by a CNN, *e.g.*, VGG-Net. We can take a certain layer of CNN as the thought vector to be fed into the decoding RNN to generate the caption.

**Application: Visual Question Answering**[18].

The visual question and answering can also be considered a special case of machine translation, where we have an encoder for image, and an encoder for question. We then concatenate the thought vectors of the image and the question, and use the concatenated thought vector to generate the answer using an RNN decoder.

**Application: Image Processing**.

For image processing such as style transfer, we can again use the encoder-decoder scheme. We can encode the input image into a thought vector. We can also encode the style as another vector. Then, we can concatenate the two thought vectors into one vector, and feed it into a decoder to generate the output image.

---

[14] Cho, Kyunghyun, et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation." *arXiv preprint arXiv:1406.1078* (2014).

[15] Cho, Kyunghyun, et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation." *arXiv preprint arXiv:1406.1078* (2014).

[16] Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." *arXiv preprint arXiv:1409.0473* (2014).

[17] Xu, Kelvin, et al. "Show, attend and tell: Neural image caption generation with visual attention." *International conference on machine learning*. PMLR, 2015.

[18] Antol, Stanislaw, et al. "VQA: Visual question answering." *Proceedings of the IEEE international conference on computer vision*. 2015.

**Memory and Attention**.

For text based Q&As, *e.g.*., answering the questions based on Wikipedia, we can encode the sentences in Wikipedia into thought vectors. These vectors serve as memory. For an input question, we can encode it into a thought vector. By matching the thought vector of the question to the thought vectors of the memory, we can decide which sentence we want to pay attention, using a soft-max probability distribution over the sentences. The weighted sum of the thought vectors of the sentences weighted by attention probabilities then becomes a combined thought vector, which, together with the thought vector of the question, is decoded into the answer.

## 4.10 Generative Adversarial Network

**Supervised Decoder**.

Let $h$ be the thought vector and let $g_\theta(h)$ be the decoder, which is a top-down CNN parametrized by $\theta$. We can learn the decoder by minimizing $\|X - g_\theta(h)\|^2$ if we have training data $\{h_i, X_i\}$.

**Jenson-Shannon Divergence**[19].

We have discussed about Kullback-Leibler divergence in Chapter 1. In previous chapters, we usually use cross entropy as the loss function, instead of Kullback-Leibler divergence, since term $E_{p(X)}[\log p(X)]$ in it is a fixed term when $p(X)$ is fixed, although we do not know the real distribution, and the rest of the term is exactly the cross entropy. When $p(X)$ is changing over time, it is not suitable to use cross entropy loss and we have to use Kullback-Leibler divergence[20]. But in practice, Kullback-Leibler divergence has some critical problems during training, such as it is asymmetric. Therefore, Jenson-Shannon divergence, *a.k.a.* JSD, is developed to guarantee the symmetry of the divergence, which is defined as

$$JSD(p_1, p_2) = KL\left(p_1 \,\middle|\, \frac{p_1 + p_2}{2}\right) + KL\left(p_2 \,\middle|\, \frac{p_1 + p_2}{2}\right)$$

Like Kullback-Leibler divergence, Jenson-Shannon divergence is also a method of measuring the similarity between two probability distributions. Jenson-Shannon divergence is symmetric and has finite value ranging in $[0,1]$ compared to Kullback-Leibler divergence, which makes it suitable in network training.

**Generative Adversarial Network**[21, 22].

We can assume a simple known prior distribution $h \sim p(h)$. Then the decoder defines a generative model. Such models are called generative models, which is a non-linear generalization of the factor analysis model.

The model can be learnt by generative adversarial networks, *a.k.a.* GAN, where we pair the generator model $G$ with a discriminator model $D$, where for an image $X$, $D(X)$ is the probability that $X$ is a true image instead of a generative image. We can train the pair of $(G, D)$ by an adversarial scheme. Specifically, let $G(h) = g_\theta(h)$ be the generator. Let

$$V(D, G) = E_{P_{data}}[\log D(X)] + E_{h\sim p(h)}\left[\log(1 - D\big(G(h)\big)\right]$$

which is the negated binary cross entropy loss, which can be regarded as the likelihood. that is, for real images in the training sample $X \sim P_{data}$, we only focus on the prediction probability of real images, *i.e.*, $D(X)$; for fake images in the generative network $G(h)$ for $h \sim p(h)$, we only focus on the prediction probability of fake images, *i.e.*, $1 - D\big(G(h)\big)$. Therefore, $V(D, G)$ is actually the negated binary cross entropy loss.

Then, we learn $D$ and $G$ by

$$\min_G \max_D V(D, G)$$

that is, the discriminator $D$ wants to maximize the likelihood, while the generator $G$ wants to fool the discriminator $D$. In practice, the training of $G$ is usually modified into maximizing $E_{h\sim p(h)}\left[\log D\big(G(h)\big)\right]$ to avoid vanishing gradient problem.

For a given $\theta$, let $p_\theta$ be the distribution of $g_\theta(h)$ with $h \sim p(h)$. A perfect discriminator according to the Bayesian rule is

$$D(x) = \frac{P_{data}(x)}{P_{data}(x) + p_\theta(x)}$$

[19] Fuglede, Bent, and Flemming Topsoe. "Jensen-Shannon divergence and Hilbert space embedding." *International Symposium on Information Theory, 2004. ISIT 2004. Proceedings.*. IEEE, 2004.

[20] https://zhuanlan.zhihu.com/p/74075915

[21] Goodfellow, Ian J., et al. "Generative adversarial networks." *arXiv preprint arXiv:1406.2661* (2014).

[22] Radford, Alec, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks." *arXiv preprint arXiv:1511.06434* (2015).

where we assume an equal number of real and fake examples, which is derived by

$$\frac{\partial V}{\partial D} = 0 \quad \Longrightarrow \quad \frac{P_{data}(X)}{D(X)} - \frac{p_\theta(X)}{1 - D(X)} = 0 \quad \Longrightarrow \quad D(X) = \frac{P_{data}(X)}{P_{data}(X) + p_\theta(X)}$$

For generator, we want to make $p_\theta(X)$ similar to $P_{data}(X)$, therefore we use Jensen-Shannon divergence to measure the similarity between two distributions, that is,

$$\begin{aligned} JSD(P_{data} \mid p_\theta) &= KL(P_{data} \mid p_{mix}) + KL(p_\theta \mid p_{mix}) \\ &= \sum_X \left[ P_{data}(X) \log \frac{P_{data}(X)}{p_{mix}(X)} + p_\theta(X) \log \frac{p_\theta(X)}{p_{mix}(X)} \right] \\ &= \sum_X \left[ P_{data}(X) \log \frac{2P_{data}(X)}{P_{data}(X) + p_\theta(X)} + p_\theta(X) \log \frac{2p_\theta(X)}{P_{data}(X) + p_\theta(X)} \right] \\ &= \sum_X \left[ P_{data}(X) \log 2D(X) + p_\theta(X) \log 2(1 - D(X)) \right] \\ &= 2 \log 2 + V(D, G) \end{aligned}$$

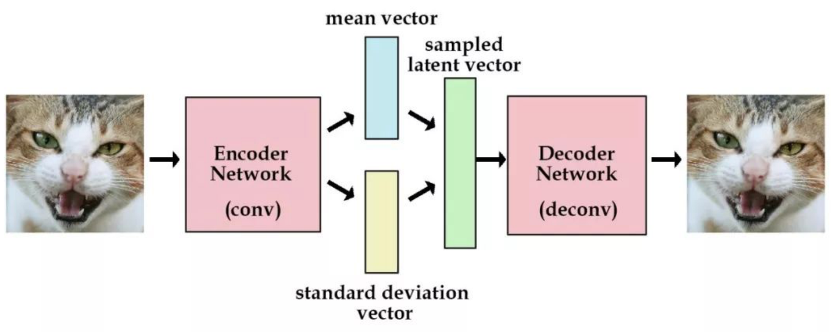
where $p_{mix} = (P_{data} + p_\theta)/2$. Notice that in the previous derivation, we use the fact that

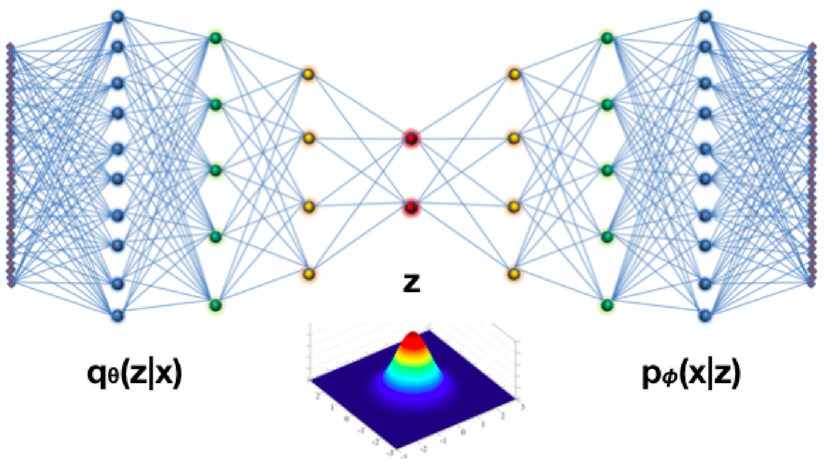
$$P_{data}(X) + p_\theta(X) = \frac{P_{data}(X)}{D(X)} = \frac{p_\theta(X)}{1 - D(X)}$$

Therefore, for generator, we want to minimize $JSD(P_{data} \mid p_\theta)$, which is equivalent with minimizing $V(D, G)$ since $2 \log 2$ is a constant.

## 4.11 Variational Auto-Encoder

Variational auto-encoder[23, 24], *a.k.a.* VAE, is generative models, akin to generative adversarial networks. Variational auto-encoder models make strong assumptions concerning the distribution of the latent variable $h$. Its main architecture is illustrated as follows, which consists of an encoder network $q_\phi(h \mid X)$ and a decoder network $p_\theta(X \mid h)$, along with a latent variable $h$.





[23] Doersch, Carl. "Tutorial on variational autoencoders." *arXiv preprint arXiv:1606.05908* (2016).
[24] Kingma, Diederik P., and Max Welling. "Auto-encoding variational bayes." *arXiv preprint arXiv:1312.6114* (2013).

**Variational Inference**[25, 26].

We want to maximize the log-likelihood of all observations, but unfortunately it is impractical to compute it directly. Recall the Jensen's inequality as applied to the probability distributions:

**Jensen's inequality** (for probability distributions). When $f$ is concave, $f(E[X]) \geq E[f(X)]$.

We use the Jensen's inequality on the log-likelihood of the observations.

$$\begin{aligned}
\log p_\theta(X) &= \log \int_h p_\theta(h, X) \mathrm{d}h \\
&= \log \int_h p_\theta(h, X) \frac{q_\phi(h \mid X)}{q_\phi(h \mid X)} \mathrm{d}h \\
&= \log \left( E_{h \sim q_\phi(h|X)} \left[ \frac{p_\theta(h, X)}{q_\phi(h \mid X)} \right] \right) \\
&\geq E_{h \sim q_\phi(h|X)} \left[ \log p_\theta(h, X) \right] - E_{h \sim q_\phi(h|X)} \left[ \log q_\phi(h \mid X) \right]
\end{aligned}$$

which gives us the formal form of **evidence lower bound (ELBO)**. We can choose a family of variational distributions, *i.e.*, a parameterization of a distribution of the latent variables, such that the expectations are computable. Then, we maximize the evidence lower bound to find the parameters that gives as tight a bound as possible on the marginal probability of $x$. Also, ELBO can be represented in the following term.

$$\begin{aligned}
\mathrm{ELBO}(X) &= E_{h \sim q_\phi(h|X)} \left[ \log p_\theta(h, X) \right] - E_{h \sim q_\phi(h|X)} \left[ \log q_\phi(h \mid X) \right] \\
&= E_{h \sim q_\phi(h|X)} \left[ \log \frac{p_\theta(h, X)}{q_\phi(h \mid X)} \right] \\
&= E_{h \sim q_\phi(h|X)} \left[ \log \frac{p(h) p_\theta(X \mid h)}{q_\phi(h \mid X)} \right] \\
&= E_{h \sim q_\phi(h|X)} \left[ \log p_\theta(X \mid h) \right] - \mathrm{KL}(q_\phi(h \mid X) \mid p(h))
\end{aligned}$$

Notice that here we omit parameter $\theta$ in $p(h)$ since latent vector is not determined by model $p_\theta$, so it is basically a prior to the variational inference model. In VAE, $h$ is assumed to follow the standard Gaussian distribution $N(0, I)$. Therefore, in order to maximize the log probability of the observations, we just need to maximize to ELBO, which is the objective of VAE since directly maximizing log likelihood is intractable:

$$\mathfrak{L}(\phi, \theta; X) = \mathrm{ELBO}(X) = E_{h \sim q_\phi(h|X)}[\log p(X \mid h)] - \mathrm{KL}\left( q_\phi(h \mid X) \,\middle|\, p(h) \right)$$

For all training data, we just need to add an outer $E_{X \sim P_{data}(X)}$ to get the objective, that is,

$$\mathfrak{L}(\phi, \theta) = \mathrm{ELBO} = E_{X \sim P_{data}(X)} L(\phi, \theta; X) = E_{X \sim P_{data}(X)} \mathrm{ELBO}(X)$$

which leads to

$$\mathfrak{L}(\phi, \theta) = E_{X \sim P_{data}(X)} \left[ E_{h \sim q_\phi(h|X)}[\log p(X \mid h)] - \mathrm{KL}\left( q_\phi(h \mid X) \,\middle|\, p_\theta(h) \right) \right]$$

**VAE Objective Optimization**.

Now, let us revisit on the KL divergence between $q_\phi(h \mid X)$ and $p_\theta(h \mid X)$.

$$\begin{aligned}
\mathrm{KL}(q_\phi(h \mid X) \mid p_\theta(h \mid X)) &= E_{X \sim P_{data}(X)} E_{h \sim q_\phi(h|X)} \left[ \log \frac{q_\phi(h \mid X)}{p_\theta(h \mid X)} \right] \\
&= E_{X \sim P_{data}(X)} E_{h \sim q_\phi(h|X)} \left[ \log \frac{q_\phi(h \mid X) p_\theta(X)}{p_\theta(h, X)} \right] \\
&= -E_{X \sim P_{data}(X)} E_{h \sim q_\phi(h|X)} \left[ \log \frac{p_\theta(h, X)}{q_\phi(h \mid X)} \right] + E_{X \sim P_{data}(X)} E_{h \sim q_\phi(h|X)} \left[ \log p_\theta(X) \right] \\
&= -\mathrm{ELBO} + E_{X \sim P_{data}(X)} \left[ \log p_\theta(X) \right]
\end{aligned}$$

[25] https://www.cs.princeton.edu/courses/archive/fall11/cos597C/lectures/variational-inference-i.pdf

[26] Odaibo, Stephen. "Tutorial: Deriving the standard variational autoencoder (vae) loss function." *arXiv preprint arXiv:1907.08956* (2019).

Therefore, VAE objective ELBO can be re-written as

$$\text{ELBO} = E_{X\sim P_{data}(X)}[\log p_\theta(X)] - \text{KL}\left( q_\phi(h \mid X) \,\middle|\, p_\theta(h \mid X) \right)$$

We can also take $-ELBO$ as VAE objective, which can be viewed as the loss function and thus we just want to minimize it. Then, we can derive the following alternative VAE objective.

$$\begin{aligned}
\max_{\theta,\phi} \mathfrak{L} &= \min_{\theta,\phi} -\text{ELBO} \\
&= \min_{\theta,\phi} \left(\text{KL}(q_\phi(h \mid X) \mid p_\theta(h \mid X)) - E_{X\sim P_{data}(X)}[\log p_\theta(X)]\right) \\
&= \min_{\theta,\phi} \left(\text{KL}(q_\phi(h \mid X) \mid p_\theta(h \mid X)) - E_{X\sim P_{data}(X)}[\log p_\theta(X)] + E_{X\sim P_{data}(X)}[P_{data}(X)]\right) \\
&= \min_{\theta,\phi} \left(\text{KL}(q_\phi(h \mid X) \mid p_\theta(h \mid X)) + E_{X\sim P_{data}(X)}\left[\log \frac{P_{data}(X)}{p_\theta(X)}\right]\right) \\
&= \min_{\theta,\phi} \left(\text{KL}(q_\phi(h \mid X) \mid p_\theta(h \mid X)) + \text{KL}(P_{data}(X) \mid p_\theta(X))\right) \\
&= \min_{\theta,\phi} \text{KL}(P_{data}(X) q_\phi(h \mid X) \mid p_\theta(h, X))
\end{aligned}$$

Therefore, if we overload $q_\phi$ as $P_{data}(X)q_\phi(h \mid X)$, and overload $p_\theta$ as $p_\theta(h, X) = p(h)p_\theta(X \mid h)$, then we just need to minimize the KL divergence between $q_\phi$ and $p_\theta$. Let $Q = \{q_\phi, \forall\phi\}, P = \{p_\theta, \forall\theta\}$. The VAE problem is to $\min_{p\in P, q\in Q} KL(q \mid p)$. Starting from $p_0$, the VAE iterates the following steps:

1. $q_{t+1} = \arg\min_{q\in Q} KL(q \mid p_t)$;
2. $p_{t+1} = \arg\min_{p\in P} KL(q_{t+1} \mid p_t)$.

which is an alternating projection, and the minimization can also be replaced by gradient descent.

The wake-sleep algorithm is the same as VAE in step 2; however in step 1, it is

$$q_{t+1} = \arg\min_{q\in Q} KL(p_t \mid q)$$

where we generate dream data from $p_t$, and learn $q_\phi(h \mid X)$ from the dream data, that is, we generate some samples from our own "belief", and have posterior to match our belief[27].

[27] https://www.comp.nus.edu.sg/~kanmy/courses/6101_1910/W4-notes.pdf