Neural networks

Ying Nian Wu and Quanshi Zhang

Contents

Note on Deep Learning and Backgrounds for M232A at the University of California, Los Angeles (UCLA).

All the figures are taken from the web.

1 Neural networks

1.1 Two-layer perceptron

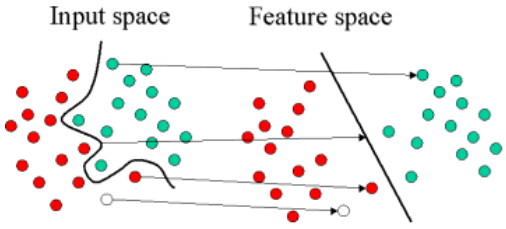


Figure 1: The positive examples and negative examples cannot be separated by a hyperplane in the original space. We can transform each X_i into a feature h_i, so that the examples can be separated by a hyperplane in the feature space.

obs	input	hidden	output
1	$X_1^\top$	$h_1^\top$	y_1
2	$X_2^\top$	$h_2^\top$	y_2
...			
n	$X_n^\top$	$h_n^\top$	y_n

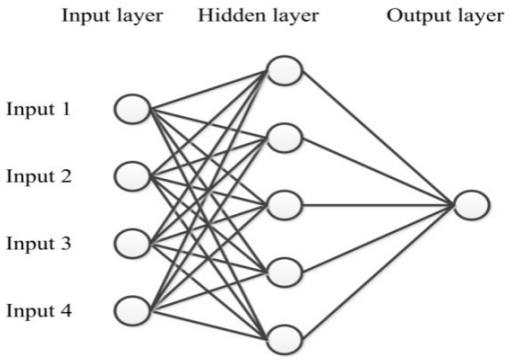


Figure 2: A two-layer feedforward neural network. The output follows a logistic regression on the hidden vector. Each component of the hidden vector in turn follows a logistic regression on the input vector.

A perceptron seeks to separate the positive examples and negative examples by projecting them onto a vector β, or in other words, separating them using a hyperplane that is perpendicular to β. If the data are not linearly separable, a perceptron cannot work. We may need to transform the original data into some features so that the features can be linearly separated. See Figure 1. One way to solve this problem is to generalize the perceptron into multi-layer perceptron. This structure is also called feedforward neural network. See Figure 2.

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

$$
\begin{aligned}
& y_i \sim \text{Bernoulli}(p_i), \\
& p_i = \text{sigmoid}(h_i^\top \beta) = \text{sigmoid}(\sum_{k=1}^{d} \beta_k h_{ik}), \\
& h_{ik} = \text{sigmoid}(X_i^\top \alpha_k) = \text{sigmoid}(\sum_{j=1}^{p} \alpha_{kj} x_{ij}).
\end{aligned}
$$

Back-propagation[19]

$$
\begin{aligned}
& P = \prod_i p_i^{y_i} (1-p_i)^{1-y_i} \\
\Longrightarrow \quad & \log P = \sum_i [y_i \log p_i + (1-y_i) \log(1-p_i)] \\
\Longrightarrow \quad & \log P = \sum_{i=1}^{n} \{y_i \{A - \log[1+\exp(A)]\} + (1-y_i)\{\log 1 - \log[1+\exp(A)]\}\} \qquad \text{where} \quad A = \sum_{k=1}^{d} \beta_k h_{ik} \\
\Longrightarrow \quad & \log P = \sum_{i=1}^{n} \{y_i A - \log[1+\exp(A)]\} \qquad \text{where} \quad A = \sum_{k=1}^{d} \beta_k h_{ik}
\end{aligned}
$$

The log-likelihood is

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

This time we use $\mathscr{L}(\beta, \alpha)$ to denote the log-likelihood function, which is to be maximized.

The gradient is

$$
\begin{aligned}
& \frac{\partial \mathscr{L}}{\partial \beta} = \sum_{i=1}^{n} (y_i - p_i) h_i, \\
& \frac{\partial \mathscr{L}}{\partial \alpha_k} = \frac{\partial \mathscr{L}}{\partial h_k} \frac{\partial h_k}{\partial \alpha_k} = \sum_{i=1}^{n} \frac{\partial \mathscr{L}}{\partial h_{ik}} \frac{\partial h_{ik}}{\partial \alpha_k} = \sum_{i=1}^{n} (y_i - p_i) \beta_k h_{ik} (1 - h_{ik}) X_i
\end{aligned}
$$

where

$$
\begin{aligned}
\frac{\partial \mathscr{L}}{\partial h_{ik}} =& \frac{\partial}{\partial h_{ik}} \sum_{i=1}^{n} \left\{ y_i \sum_{k=1}^{d} \beta_k h_{ik} - \log \left[1 + \exp \left(\sum_{k=1}^{d} \beta_k h_{ik} \right) \right] \right\} \\
=& y_i \beta_k - \frac{\exp\left(\sum_{k=1}^{d} \beta_k h_{ik}\right) \beta_k}{1 + \exp\left(\sum_{k=1}^{d} \beta_k h_{ik}\right)} \\
=& \left\{ y_i - \frac{\exp\left(\sum_{k=1}^{d} \beta_k h_{ik}\right)}{1 + \exp\left(\sum_{k=1}^{d} \beta_k h_{ik}\right)} \right\} \beta_k \\
=& \left\{ y_i - \frac{1}{1 + \exp\left(-\sum_{k=1}^{d} \beta_k h_{ik}\right)} \right\} \beta_k \\
=& (y_i - p_i) \beta_k \\
\frac{\partial h_{ik}}{\partial \alpha_k} =& \frac{\partial}{\partial \alpha_k} \left(\frac{1}{1 + \exp(-\alpha_k^\top X_i)} \right) \\
=& \frac{\exp(-\alpha_k^\top X_i) X_i}{[1 + \exp(-\alpha_k^\top X_i)]^2} \\
=& \left(\frac{1}{1 + \exp(-\alpha_k^\top X_i)} \right) \left(\frac{\exp(-\alpha_k^\top X_i)}{1 + \exp(-\alpha_k^\top X_i)} \right) X_i \\
=& h_{ik}(1 - h_{ik}) X_i
\end{aligned}
$$

$\partial \mathscr{L} / \partial \alpha_k$ is calculated by chain rule. Again the gradient descent learning algorithm learns from mistake or error $y_i - p_i$. The chain rule back-propagates the error to assign the blame on β and α in order for β and α to update. If the current network makes a mistake on the i-th example, β will change to be more aligned with h_i, while each α_k also changes to be more aligned with X_i. The amount of change depends on β_k as well as $h_{ik}(1 - h_{ik})$, which measures how big a role played by $X^\top \alpha_k$ in predicting y_i. If $X^\top \alpha_k$ plays a big role, then α_k should receive the blame and make change.

Rectified linear unit (ReLU)[17, 16]

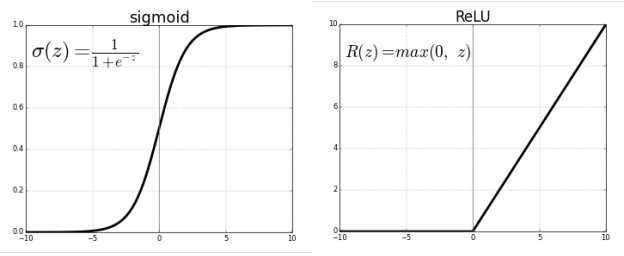


Figure 3: The left is the sigmoid function, and the right is the ReLU function. The sigmoid function saturates at the two ends, causing the gradient to vanish. The ReLU does not saturate for big positive input.

In modern neural net, the non-linearity is often the rectified linear unit (ReLU) $\max(0,a)$, see Figure 3.

$$
\begin{aligned}
& y_i \sim \text{Bernoulli}(p_i), \\
& p_i = \text{sigmoid}(h_i^\top \beta) = \text{sigmoid}\left(\sum_{k=1}^{d} \beta_k h_{ik}\right), \\
& h_{ik} = \max(X_i^\top \alpha_k, 0) = \max\left(\sum_{j=1}^{p} \alpha_{kj} x_{ij}, 0\right).
\end{aligned}
$$

For ReLU $\max(0,a)$, we should replace $h_{ik}(1-h_{ik})$ in the back-propagation by $\mathbf{1}(h_{ik}>0)$, which is a binary detector.

$$
\frac{\partial h_{ik}}{\partial \alpha_k} = \mathbf{1}(\alpha_k^\top X_i > 0) \cdot X_i = \mathbf{1}(h_{ik} > 0) \cdot X_i
$$

Relationships with the sigmoid function: The sigmoid function saturates at the two ends, where the derivatives are close to zero. This can cause the vanishing gradient problem in back-propagation, so that the network cannot learn from the error. The ReLU function does not saturate for big positive input, which indicates the existence of a certain patten. This helps avoid the vanishing gradient problem.

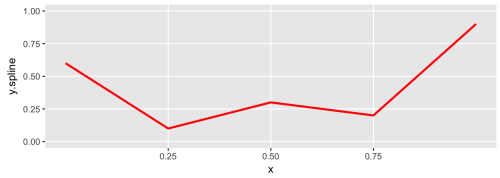


Figure 4: Spline is a continuous piecewise linear function, where at each knot, the spline makes a turn by changing the slope. The neural net can be viewed a high dimensional spline with exponentially many linear pieces.

Relationships with the linear spline model: Recall the linear spline model $f(x_i) = \beta_0 + \sum_{j=1}^{p} \beta_j \max(0, x_i - a_j)$, see Figure 4, where a_j are the knots, where the spline makes a turn, i.e., a change of slope by β_j. The neural net with ReLU can be viewed a high-dimensional spline, or piecewise linear mapping. If there are many layers in the neural net, 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.

1.2 Multi-layer network

See Figure 5 for an illustration of multi-layer perceptron or a feedforward neural network. It consists 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 it by a weight matrix, plus a bias vector, and then transforming each element of the resulting vector by sigmoid or ReLU etc.

More formally, the network has the following recursive structure:

$$
\begin{aligned}
h_l &= f_l(s_l), \\
s_l &= W_l h_{l-1} + b_l,
\end{aligned}
$$

for $l = 1, \ldots, L$, where l denotes the layer, with $h_0 = X$, which is the input vector, and h_L is used to predict Y, which is the output, based on a log-likelihood function $L(Y, h_L) = \sum_i \log p(y_i | X_i)$. Here h_L corresponds to

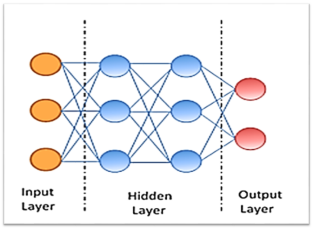


Figure 5: Multi-layer perceptron (two hidden layers here) or feedforward neural network. There is an input layer (or an input vector), an output layer, and multiple hidden layers (or hidden vectors). Each layer is a linear transformation of the layer beneath, followed by an element-wise non-linear transformation.

$X^\top\beta$ in the previous section. W_l and b_l are the weight matrix and bias vector respectively. f_l is element-wise transformation, i.e., each component of h_l is obtained by a non-linear transformation of the corresponding component of s_l, i.e., $h_{lk} = f_l(s_{lk})$ for each k.

Chain rule in matrix form

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

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

To understand the notation, we can treat $\partial Y = (\partial y_i, i = 1,...,m)^\top$ as a column vector, and $1/\partial X = (1/\partial x_j, j = 1,...,n)^\top$ as another column vector. Now we have two vectors of operations, instead of numbers. The product of the elements of the two vectors is understood as composition of the two operators, i.e., $\partial y_i(1/\partial x_j) = \partial y_i/\partial x_j$. Then $\partial Y/\partial X^\top$ is a squared matrix according to the matrix multiplication rule.

If $Y = AX$, then $y_i = \sum_k a_{ik}x_k$. Thus $\partial y_i/\partial x_j = a_{ij}$. So $\partial Y/\partial X^\top = A$.

If $Y = X^\top SX$, where S is symmetric, then $\partial Y/\partial X = 2SX$.

If $S = I$, $Y = \|X\|^2$, $\partial Y/\partial X = 2X$.

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

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

Thus

$$\frac{\partial Y}{\partial Z^\top} = \frac{\partial Y}{\partial X^\top}\frac{\partial X}{\partial Z^\top}.$$

Multi-layer back-propagation

For multi-layer network, the chain rule is such that

$$\frac{\partial L}{\partial s_l^\top} = \frac{\partial L}{\partial h_l^\top}\frac{\partial h_l}{\partial s_l^\top} = \frac{\partial L}{\partial h_l^\top}f_l',$$

$$\frac{\partial L}{\partial h_{l-1}^\top} = \frac{\partial L}{\partial h_l^\top}\frac{\partial h_l}{\partial s_l^\top}\frac{\partial s_l}{\partial h_{l-1}^\top} = \frac{\partial L}{\partial h_l^\top}f_l'W_l,$$

where f_l' is a diagonal matrix because f_l is element-wise, i.e., the k-th diagonal element of f_l' is $\partial h_{lk}/\partial s_{lk}$, which is $h_{lk}(1-h_{lk})$ for sigmoid, and $1(h_{lk}>0)$ for ReLU.

$$W_l = \frac{\partial s_l}{\partial h_{l-1}^\top} = \begin{bmatrix} \frac{\partial s_{l,1}}{\partial h_{l-1,1}} & \frac{\partial s_{l,1}}{\partial h_{l-1,2}} & \frac{\partial s_{l,1}}{\partial h_{l-1,3}} & \frac{\partial s_{l,1}}{\partial h_{l-1,4}} & \frac{\partial s_{l,1}}{\partial h_{l-1,5}} \\ \frac{\partial s_{l,2}}{\partial h_{l-1,1}} & \frac{\partial s_{l,2}}{\partial h_{l-1,2}} & \frac{\partial s_{l,2}}{\partial h_{l-1,3}} & \frac{\partial s_{l,2}}{\partial h_{l-1,4}} & \frac{\partial s_{l,2}}{\partial h_{l-1,5}} \\ \frac{\partial s_{l,3}}{\partial h_{l-1,1}} & \frac{\partial s_{l,3}}{\partial h_{l-1,2}} & \frac{\partial s_{l,3}}{\partial h_{l-1,3}} & \frac{\partial s_{l,3}}{\partial h_{l-1,4}} & \frac{\partial s_{l,3}}{\partial h_{l-1,5}} \\ \frac{\partial s_{l,4}}{\partial h_{l-1,1}} & \frac{\partial s_{l,4}}{\partial h_{l-1,2}} & \frac{\partial s_{l,4}}{\partial h_{l-1,3}} & \frac{\partial s_{l,4}}{\partial h_{l-1,4}} & \frac{\partial s_{l,4}}{\partial h_{l-1,5}} \\ \frac{\partial s_{l,5}}{\partial h_{l-1,1}} & \frac{\partial s_{l,5}}{\partial h_{l-1,2}} & \frac{\partial s_{l,5}}{\partial h_{l-1,3}} & \frac{\partial s_{l,5}}{\partial h_{l-1,4}} & \frac{\partial s_{l,5}}{\partial h_{l-1,5}} \end{bmatrix}$$

$$f_l' = \frac{\partial h_l}{\partial s_l^\top} = \begin{bmatrix} \frac{\partial h_{l1}}{\partial s_{l1}} & 0 & 0 & 0 & 0 \\ 0 & \frac{\partial h_{l2}}{\partial s_{l2}} & 0 & 0 & 0 \\ 0 & 0 & \frac{\partial h_{l3}}{\partial s_{l3}} & 0 & 0 \\ 0 & 0 & 0 & \frac{\partial h_{l4}}{\partial s_{l4}} & 0 \\ 0 & 0 & 0 & 0 & \frac{\partial h_{l5}}{\partial s_{l5}} \end{bmatrix}$$

Transpose the above equation,

$$\frac{\partial L}{\partial h_{l-1}} = W_l^\top f_l' \frac{\partial L}{\partial h_l}.$$

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

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

Combining all the rows,

$$\left(\frac{\partial L}{\partial W_l}\right)_{K\times K} = \left(\frac{\partial L}{\partial s_l}\right)_{K\times 1} \left(h_{l-1}^\top\right)_{1\times K} = \left(f_l'\right)_{K\times K} \left(\frac{\partial L}{\partial h_l}\right)_{K\times 1} \left(h_{l-1}^\top\right)_{1\times K}$$

Define

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

we have

$$\Delta h_{l-1} = W_l^\top D_l \Delta h_l,$$
$$\Delta W_l = D_l \Delta h_l h_{l-1}^\top.$$

We can also include b_l by adding an intercept term to h_{l-1},

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

We can learn $\theta = (W_l, b_l, l = 1, ..., L)$ by gradient descent, which is again learning from the mistakes by error back-propagation.

1.3 Stochastic gradient descent (SGD)

Mini-batch

Let the training data be (X_i, y_i), $i = 1, ..., n$. Let $L_i(\theta) = L(y_i, X_i; \theta)$ be the loss caused by (X_i, y_i). For regression, $L(y_i, X_i; \theta) = (y_i - f(X_i))^2$, where $f(X_i)$ is parametrized by a neural network with parameters θ. For classification, $L(y_i, X_i; \theta) = -\log p(y_i|X_i)$ where $p(y_i|X_i)$ is modeled by a neural network with parameters θ, with a softmax layer at the top. Let $\mathscr{L}(\theta) = \frac{1}{n}\sum_{i=1}^n L_i(\theta)$ be the overall loss averaged over the whole training dataset. The gradient descent is

$$\theta_{t+1} = \theta_t - \eta \mathscr{L}'(\theta_t),$$

where η is the step size or learning rate, and $\mathscr{L}'(\theta)$ is the gradient.

The above gradient descent may be time consuming because we need to compute $\mathscr{L}'(\theta) = \frac{1}{n}\sum_{i=1}^n L_i'(\theta)$ by summing over all the examples. If the number of examples is large, the computation can be time consuming. We may use the following stochastic gradient descent algorithm. At each step, we randomly select i from $\{1, 2, ..., n\}$. Then we update θ by

$$\theta_{t+1} = \theta_t - \eta_t L_i'(\theta_t),$$

where η_t is the step size or learning rate, and $L_i'(\theta_t)$ is the gradient only for the i-th example. Because i is randomly selected, the above algorithm is called the stochastic gradient descent algorithm.

Instead of randomly selecting a single example, we may randomly select a mini-batch, and replace $L_i'(\theta_t)$ by the average of this mini-batch.

According to the Robbins-Monroe theory for stochastic approximation, we usually need the following conditions to ensure the algorithm to converge to a local minimum. (1) $\sum_{t=1}^\infty \eta_t = \infty$. (2) $\sum_{t=1}^\infty \eta_t^2 < \infty$. The first condition ensures that the algorithm can go the distance toward the minimum. The second condition ensures that the algorithm will not run away from the local minimum once it arrives. One simple example is $\eta_t = c/t$ for a constant c. In practice, we need more sophisticated scheme for η_t. For instance, reducing η_t after a certain number of steps, or reducing η_t if the training error stops decrease.

Momentum, Adagrad, RMSprop, Adam

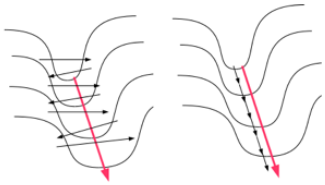


Figure 6: Momentum. The left figure illustrates the original gradient descent algorithm. The black arrow is the gradient direction, and the red arrow is the preferred direction. The right figure illustrates the gradient descent algorithm with momentum, which is along the red arrow.

The gradient descent algorithm goes downhills in the steepest direction in each step. However, the steepest direction may not be the best direction, as illustrated by Figure 6. In the left figure, the black arrows are the gradient direction. The red arrows are the preferred direction, which is the direction of momentum. It is better to move along the direction of the momentum, as illustrated by the right figure. We want to

accumulate the momentum, and let it guide the descent. The following is the stochastic gradient descent with momentum[19, 21]:

$$v_t = \gamma v_{t-1} + \eta_t g_t,$$
$$\theta_t = \theta_{t-1} - v_t.$$

where g_t is the average gradient computed from the current mini-batch, and v_t is the momentum or velocity. γ is usually set at 0.9, for accumulating the momentum, and θ is updated based on the momentum.

Adagrad modifies the gradient descent algorithm in another direction. The magnitudes of the components of g_t may be very uneven, and we need to be adaptive to that. The Adagrad[5] let

$$G_t = G_{t-1} + g_t^2,$$
$$\theta_{t+1} = \theta_t - \eta_t \frac{g_t}{\sqrt{G_t + \varepsilon}},$$

where ε is a small number to avoid dividing by 0. In the above formula, g_t^2 and $g_t/\sqrt{G_t+\varepsilon}$ denote component-wise square and division.

In Adagrad, G_t is the sum over all the time steps. It is better to sum over the recent time steps. Adadelta and RMSprop[24] use the following scheme:

$$G_t \quad = \quad \beta G_{t-1} + (1-\beta) g_t^2,$$

where β can be set at 0.9. It can be shown that

$$G_t = (1-\beta)(\beta^{t-1} g_1^2 + \beta^{t-2} g_2^2 + ... + \beta g_{t-1}^2 + g_t^2),$$

which is a sum over time with decaying weights.

The Adam[11] optimizer combines the idea of RMSprop and the idea of momentum.

$$v_t = \gamma v_{t-1} + (1-\gamma) g_t,$$
$$G_t = \beta G_{t-1} + (1-\beta) g_t^2,$$
$$v_t \leftarrow v_t/(1-\gamma), G_t \leftarrow G_t/(1-\beta),$$
$$\theta_{t+1} = \theta_t - \eta_t \frac{v_t}{\sqrt{G_t + \varepsilon}}.$$

2 Convolutional neural networks (CNN ≡ ConvNet)

2.1 Convolution, kernels, filters

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 convolutional neural network[6, 15], where W_l and b_l have a convolutional structure.

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 transformation.

$$s = W \otimes h + b$$
$$s_{uvw} = \sum_{i=1}^{R} \sum_{j=1}^{R} \sum_{k=1}^{C} W_{ijk}^{w} h_{t(u-1)-p+i, t(v-1)-p+j, k} + b^w$$

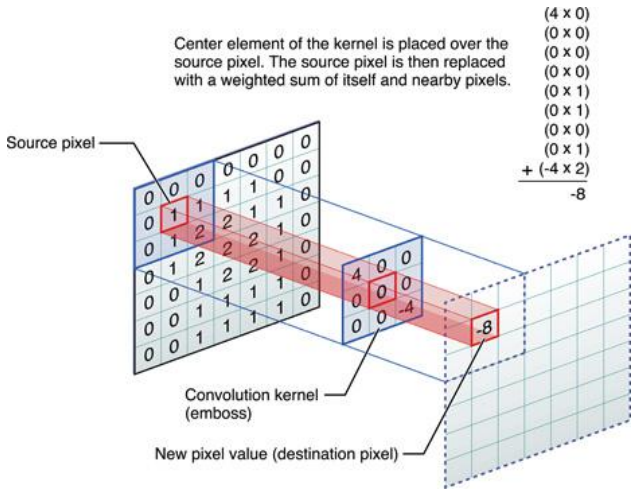


Figure 7: Local weighted summation. Here the filter or kernel is 3×3. It slides over the whole input image or feature map. At each pixel, we compute the weighted sum of the 3×3 patch of the input image, where the weights are given by the filter or kernel. This gives us an output image or filtered image or feature map.

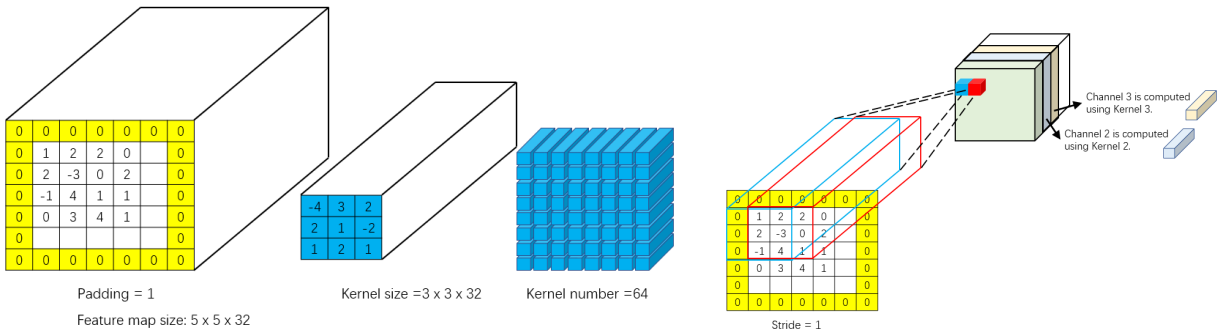


Figure 8: Padding, feature map size, kernel size, kernel number, stride.

where the kernel size is $R \times R \times C$; t denotes the stride; p denotes the padding. W^w_{ijk} denotes the unit (i, j, k) of the w-th kernel.

Figure 7 explains the local weighted summation. We convolve an input feature map with a 3×3 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 values of the 3×3 patch of the input feature map around this pixel. We apply the weighted summation around each pixel with the same weights to obtain the output 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. Then we will get different output feature maps. Each output feature map corresponds to one filter. See Figure 9 for illustration.

After the weighted summation, we may also add a bias and apply a non-linear transformation such as sigmoid or ReLU. The filter becomes non-linear.

ReLU layer:

$$h = \max\{s, \mathbf{0}\}$$

$$\text{i.e.} \qquad h_{ijk} = \max\{s_{ijk}, 0\}$$

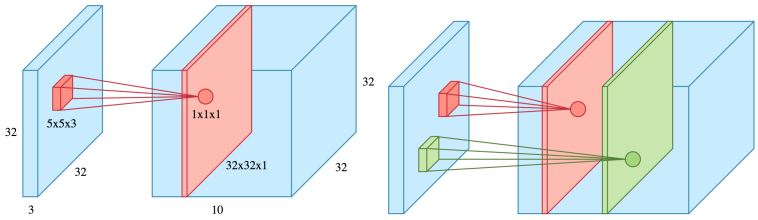


Figure 9: Convolution. The input may consist of multiple channels, illustrated by a rectangle box. Each input feature map is a slice of the box. The local weighted summation in convolution is also over the channels. If the spatial range of the filter is 3×3 and the input has 3 channels, then the filter or kernel is $3 \times 3 \times 3$ box. The spatial range can also be 1×1, then the filter involves weighted summation of the 3 channels at the same pixel. For the input image, there are 3 channels corresponding to 3 colors: red, green, black. For the hidden layers, each layer may consist of hundreds of channels. Each channel is obtained by a filter. For instance, in the figure on the right, the red feature map is obtained by the red filter, and the green feature map is obtained by the green filter.

After obtaining the feature maps in h_l, we may perform max-pooling, e.g., for each feature map, at each pixel, we replace the value of this pixel by the maximum of the 3×3 patch around this pixel. We may also do average pooling, i.e., at each pixel, we replace the value of this pixel by the average of the 3×3 patch around this pixel.

The mapping from h_{l-1} to h_l may also involve sub-sampling to reduce the size of feature maps. For instance, after obtaining a feature map by a filter, we can partition the feature map into 2×2 blocks, and within each block, we only keep the upper left pixel. This will reduce the width and height of the feature map by half. h_{l-1} may have more channels than h_l, because each element of h_l covers a bigger spatial extent than each element of h_{l-1}, and there are more patterns of bigger spatial extent.

The output feature map may also be 1×1, whose value is a weighted sum of all the elements in h_{l-1}. h_l may consists of a number of such 1×1 maps. It is called the fully connected layer because each element of h_l is connected to all the elements in h_{l-1}.

Fully connected layers are special convolutional layers.

A convolutional network consists of multiple layers of convolutional and fully connected layers, with max pooling and sub-sampling between the layers. See Figure 10. The network can be learned by back-propagation.

2.2 Softmax layer for classification

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 $h_i = (h_{ik}, k = 1, ..., K)$. Then we output the probability that the input X_i belongs to category k

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

We use the notation k' to avoid the confusion with the index k in the numerator. This is called soft-max probability. In testing, we can simply classify X_i to category k whose h_{ik} is the maximum in h_i. This is hard max.

Suppose our training dataset is (X_i, y_i), $i = 1, ..., n$, where X_i is the input image, and y_i is the output category. Let $p(y|X, \theta)$ be the probability that the input image X belongs to the category y according to the

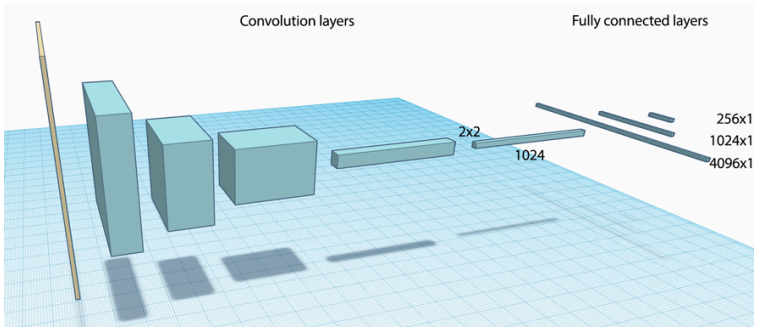


Figure 10: Convolutional neural network (CNN or ConvNet). The input image has 3 channels (R, G, B). Each subsequent layer consists of multiple channels, and is illustrated by a box. Sub-sampling is performed after some layers, so that the the box at the higher layers is smaller than the box at the lower layer in the spatial extent. Meanwhile, the box at the higher layer may have more channels than the box at the lower layer, illustrated by the fact that the box at the higher layer is longer than the box at the lower layer. A fully connected layer consists of 1×1 feature maps, and is illustrated by a horizontal line.

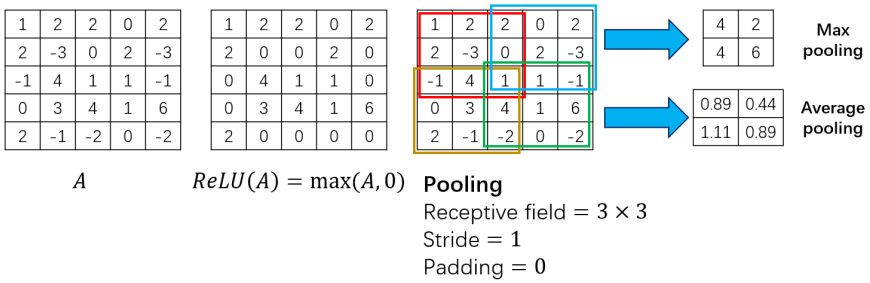


Figure 11: ReLU, max-Poolng, and avg-Pooling operations

above softmax probability. Then we can estimate $\theta = (W_l, b_l, l = 1, ..., L)$ by maximizing the log-likelihood

$$\sum_{i=1}^{n} \log p(y_i | X_i, \theta)$$

using gradient descent based on the back-propagation computation. We can define the loss to be the negative log-likelihood

$$\mathscr{L}(\theta) = \sum_{i=1}^{n} L(y_i, h_i), = -\sum_{i=1}^{n} \log p(y_i | X_i, \theta).$$

If $y_i = k$, then

$$\begin{aligned} \frac{\partial \log p(y_i | X_i, \theta)}{\partial h_{ik'}} &= 1(k' = k) - \frac{\partial}{\partial h_{ik'}} \log \sum_{k'=1}^{K} e^{h_{ik'}} \\ &= 1(k' = k) - \frac{e^{h_{ik'}}}{\sum_{k''=1}^{K} e^{h_{ik''}}} \\ &= 1(k' = k) - p_{ik'}. \end{aligned}$$

Let Y_i be the one-hot vector, i.e., if $y_i = k$, then the k-th element of Y_i is 1, and 0 otherwise. Let $p_i = (p_{ik}, k =$

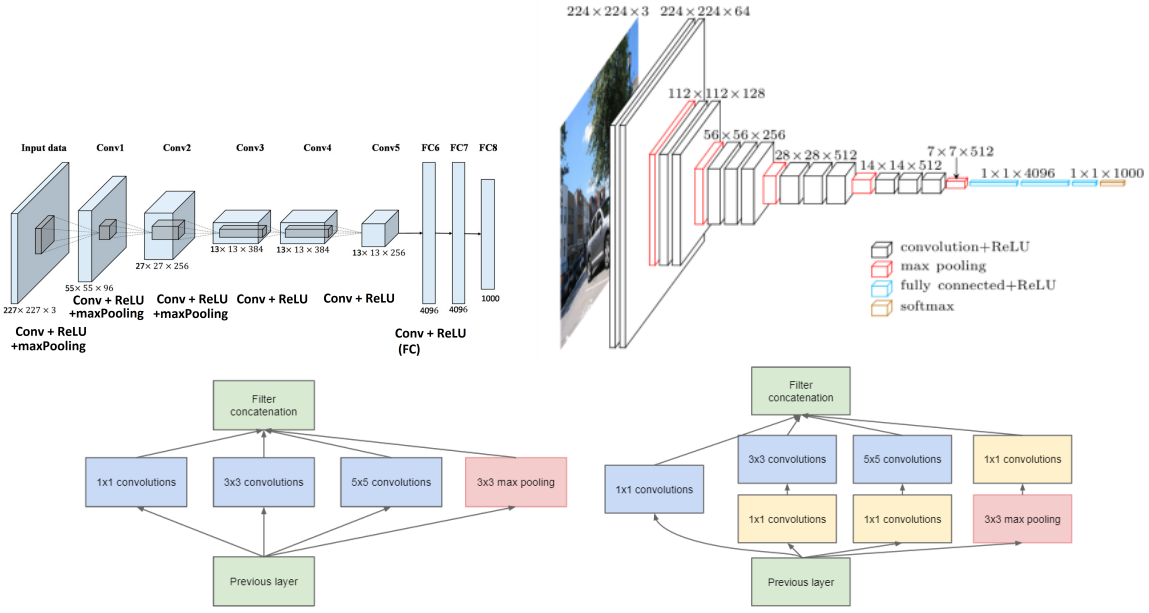
$1, ..., K)$ be the vector of predicted probabilities of the K categories. Then

$$\frac{\partial \log p(y_i|X_i, \theta)}{\partial h_i} = Y_i - p_i = e_i \qquad \text{Learn from errors}$$

We can then 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 softmax layer is related to the least square regression between Y and p. For regression, the least squares loss is $L(y_i, h_i) = \|y_i - h_i\|^2$, and the derivative with respect to h_i is $-2(y_i - h_i) = -2e_i$. Both y_i and h_i can be vectors.

2.3 Alex net, VGG net, inception net



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

Figure 12: AlexNet, VGG-16, and Inception net

The Alex net[13] is the neural network that achieved the initial breakthrough 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 softmax output layer.

The VGG net[20] is an improvement on the Alex net. There are two versions, VGG16 and VGG19, which consist of 16 hidden layers and 19 hidden layers respectively. The VGG19 has 144 million parameters. The filters of the VGG nets are all 3×3. Like Alex net, it has three fully connected layers, plus a softmax output layer.

The inception net[22, 10, 23] took its name from the movie "Inception," which has a line "we need to go deeper." The network makes extensive use of 1×1 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 1×1 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 1×1, 3×3 and 5×5, as well as max pooling.

2.4 Batch normalization layer[10]

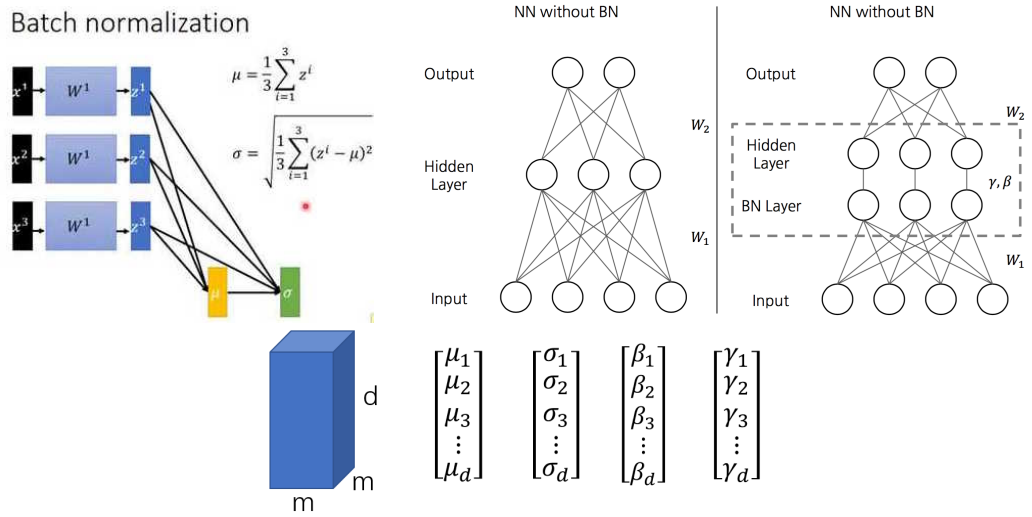


Figure 13: Batch normalization. Suppose we have a batch of 3 examples. For each element of each layer, we compute the mean μ and standard deviation σ by pooling over the 3 examples. We then normalize the element for each example, followed by a linear transformation to be learned from the data. In back-propagation, we need to treat μ and σ as a layer and compute the derivatives of μ and σ with respect to the whole batch. It is as if the whole batch becomes a single example because μ and σ are computed from this whole batch.

When training the neural net by back-propagation, the distribution of h_l keeps changing because the parameters keep changing. This may cause problem in training. We can stablize 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 instance, x is 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, we let x be an element of h_{l-1}, and we apply the batch normalization for each element of h_{l-1}. Suppose we have a batch of n training examples, so that we have $\{x_i, i = 1, ..., n\}$. The batch normalization layer is defined as follows:

$$\begin{aligned}
\mu &= \frac{1}{n}\sum_{i=1}^{n} x_i; \qquad \mu_d = \frac{1}{n}\sum_{i=1}^{n} x_{id}; \qquad \text{with channels } d = 1, 2, \ldots, D \\
\sigma_d^2 &= \frac{1}{n}\sum_{i=1}^{n}(x_{id} - \mu_d)^2; \\
\hat{x}_{id} &= \frac{x_{id} - \mu_d}{\sigma_d}; \\
y_{id} &= \beta_d + \gamma_d \hat{x}_{id}.
\end{aligned}$$

This way, we stabilize the distribution of each element of h_{l-1} during the training process. See Figure 13 for an illustration.

Note that in the batch normalization layer, there are parameters β and γ to be learned. More importantly, μ and σ^2 are functions of the whole batch. When we do chain rule calculations for back-propagation, we need to compute the derivatives of μ and σ^2 with respect to all the $(x_i, i = 1, ..., n)$. It is as if the whole batch becomes a single training example.

When we add a batch normalization layer to a CNN, we need to learn β, γ, μ and σ^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 compute d sets of β, γ, μ and σ^2 for d channels. In particular, we need to average over nm^2 numbers to compute μ and σ^2.

2.5 Residual net[8]

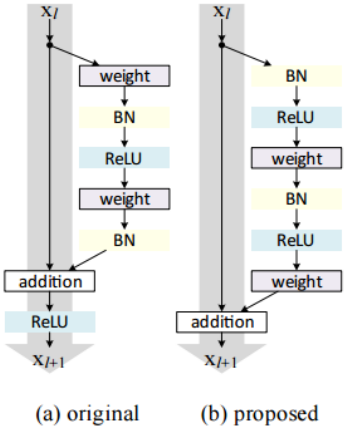


Figure 14: Residual net. The figure on the left is the original version of a residual block. The figure on the right is the revised version, which is in the form of $x_{l+1} = x_l + F(x_l)$, where F consists of two rounds of weighted sum, batch normalization, and ReLU.

A residual block in the residual net is as follows. Let x_l be the input to the residual block. Let x_{l+1} be the output of 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

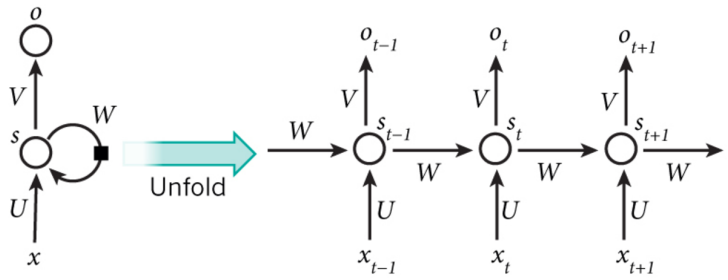
$$x_{l+1} = x_l + F(x_l),$$

as illustrated by the right plot of Figure 14. $F(x_l)$ 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.

(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 much easier for stochastic gradient descent to find a good $F(x_l)$ in the residual form than in the original form.

(2) The mapping $x_{l+1} = x_l + F(x_l)$ may model an iteration of an iterative algorithm or dynamic process, where l actually denotes the time step t instead of the real layer l. The mapping 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 simple 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 the Taylor expansion.



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

Figure 15: Recurrent neural network. The latent vector summarizes all the information about the past.

3 Recurrent neural networks

3.1 RNN

Figure 15 shows the vanilla version of the recurrent neural network [14], which can be written as

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

where we use U, V, W to denote weight matrices, and f and g represent element-wise non-linear transformations. They are be different at each occurrence.

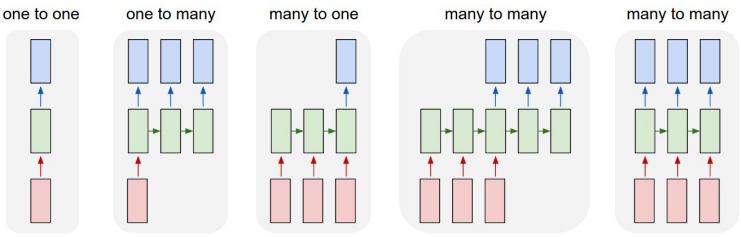


Figure 16: Different forms of inputs and outputs in RNN.

3.2 LSTM

The training of RNN is again based on back-propagation, except that we need to back-propagate over time. There is vanishing (or exploding) gradient problem. The LSTM[9] was designed to overcome this problem, by introducing memory cells.

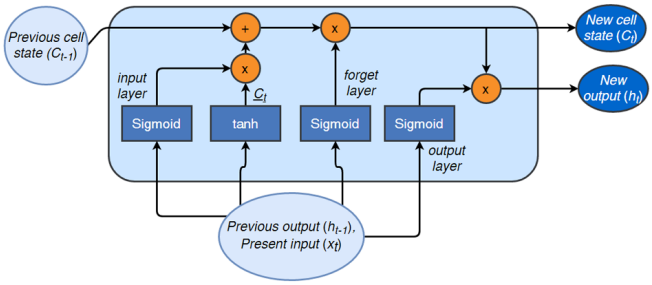


Figure 1. Architecture in a LSTM network

Figure 17: Long short term memory. There is a memory vector, and three gates, forget, input, and output.

Figure 17 illustrates the architecture of LSTM, which can be written as

$$\begin{aligned}
\Delta c_t &= f(W_c(h_{t-1}, x_t)), && \text{Representation of input information} \\
i_t &= f(W_i(h_{t-1}, x_t)), && \text{Input gate} \\
f_t &= f(W_f(h_{t-1}, x_t)), && \text{Forget gate} \\
c_t &= c_{t-1} f_t + \Delta c_t i_t, && \text{Cell state vector} \\
o_t &= f(W_o(h_{t-1}, x_t)), && \text{Output gate} \\
h_t &= o_t f(c_t), && \text{Hidden state vector} \\
y_t &= f(W h_t).
\end{aligned}$$

The gates are like if-then statements in a computer program. They are made continuous by sigmoid f so that they are differentiable. The residual net is a special case of LSTM.

3.3 GRU

The gated recurrent unit is a simplification of LSTM, by merging c_t and h_t:

$$\begin{aligned}
z_t &= f(W_z(h_{t-1}, x_t)), && \text{Update gate} \\
r_t &= f(W_r(h_{t-1}, x_t)), && \text{Reset gate} \\
\tilde{h}_t &= f(W(x_t, r_t h_{t-1})), && \text{New information from the input} \\
h_t &= (1 - z_t) h_{t-1} + z_t \tilde{h}_t, && \text{Hidden state} \\
y_t &= f(W_h h_t).
\end{aligned}$$

3.4 Encoder-decoder, thought vector

Figure 18 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, and the hidden vector is used to generate the next letter by softmax probabilities. We can think of $x_t \to h_t$ as the encoder, and $h_t \to y_t$ as the decoder[3]. 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: input $\to$ thought vector $\to$ output.

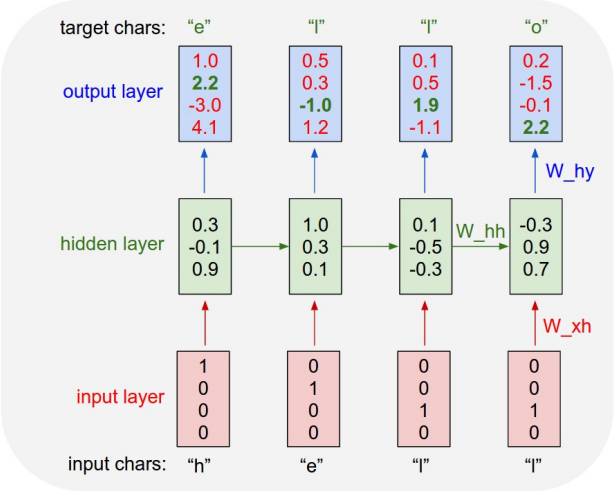


Figure 18: An RNN for sequence generation.

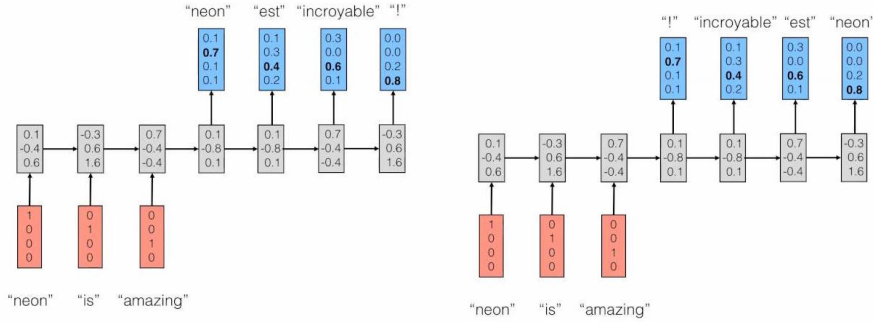


Figure 19: Machine translation. The encoding RNN encodes the input sentence into a thought vector, and the decoding RNN decodes the thought vector into an output sentence.

Translation

Figure 19 shows the RNN for machine translation[2]. 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.

Image captioning

Image captioning[25] can be considered a special case of machine translation, where the encoding RNN is replaced by a CNN (e.g., VGG). We can take a certain layer of CNN as the thought vector to be fed into the decoding RNN to generate the caption.

VQA

The visual question and answering[1] 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 question, and use the concatenated thought vector to generate the answer.

Memory and attention

For text based QA, e.g., answering 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 softmax probability distribution over the sentences. The weighted sum of the thought vectors of the sentences weighted by the attention probabilities then becomes a combined thought vector, which, together with the thought vector of the question, is decoded into the answer.

4 Generator and GAN

4.1 Encoder and decoder again

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

4.2 Supervised decoder

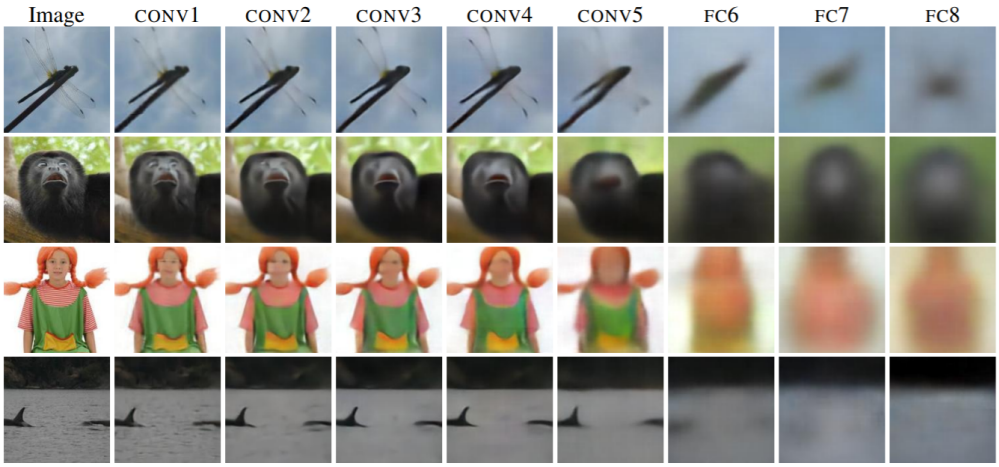


Figure 20: Image generating based on features of different intermediate layers of a CNN. $X = g_\theta(h)$ (i.e., $\hat{I} = decoder(f)$).

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

4.3 GAN

We can also assume a simple known prior distribution on h, $h \sim p(h)$. Then the decoder defines a generative model. We call such a model a generator model, which is a non-linear generalization of the factor analysis model.

The model can be learned by generative adversarial networks (GAN)[7, 18], 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

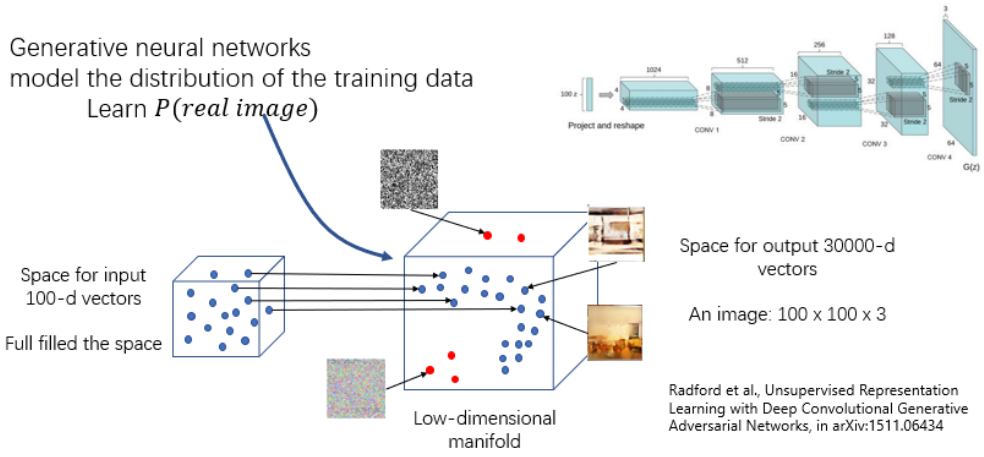


Figure 21: GAN.

instead of a generated image. We can train the pair of (G,D) by an adversarial scheme. Specifically, let $G(h) = g_\theta(h)$ be a generator. Let

$$V(D,G) = \mathrm{E}_{P_{\text{data}}}[\log p_D(X)] + \mathrm{E}_{h\sim p(h)}[\log(1-p_D(G(h))] = \mathrm{E}_{P_{\text{data}}}[\log D(X)] + \mathrm{E}_{h\sim p(h)}[\log(1-D(G(h))].$$

We learn D and G by $\min_G \max_D V(D,G)$. In practice, the training of G is usually modified into maximizing $\mathrm{E}_{h\sim p(h)}[\log D(G(h))]$ to avoid vanishing gradient problem.

For a given θ, let p_θ be the distribution of $g_\theta(h)$ with $h \sim p(h)$. A perfect discriminator according to the Bayes rule is $D(x) = P_{\text{data}}(x)/(P_{\text{data}}(x) + p_\theta(x))$ (assuming equal numbers of real and fake examples).

$$\begin{aligned}
&\frac{\partial V}{\partial D} = 0 \\
\Rightarrow &\frac{P_{\text{data}}(X)}{D(X)} - \frac{p_\theta(X)}{1-D(X)} = 0 \\
\Rightarrow &-p_\theta(X)D(X) + P_{\text{data}}(X) - P_{\text{data}}(X)D(X) = 0 \\
\Rightarrow &D(X) = \frac{P_{\text{data}}(X)}{p_\theta(X) + P_{\text{data}}(X)}
\end{aligned}$$

Then θ minimizes Jensen-Shannon

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

where $p_{\text{mix}} = (P_{\text{data}} + p_\theta)/2$.

4.4 Geometry of VAE

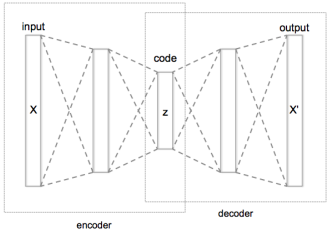


Figure 22: VAE. We use h to denote the code z in the figure.

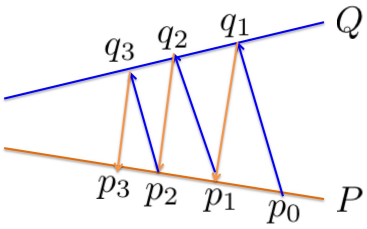


Figure 23: VAE as alternating projection.

The loss function of a VAE[12, 4] is given as

$$L(\phi, \theta, X) = \mathrm{KL}(q_\phi(h|X)|p_\theta(h|X)) - \mathrm{E}_{q_\phi(h|X)}(\log p_\theta(X|h))$$

where people usually use a multivariate Gaussian $p(h) = N(\mathbf{0}, \mathbf{I})$ as a prior to represent $p_\theta(h|X)$. We can consider $q_\phi(h|X)$ as an encoder network and consider $p_\theta(X|h)$ as a decoder network.

Let $q_\phi(h|X)$ be the variational inference model to approximate the posterior $p_\theta(h|X)$, the VAE objective can be also rewritten as follows

$$\mathrm{KL}(P_{\text{data}}(X)|p_\theta(X)) + \mathrm{KL}(q_\phi(h|X)|p_\theta(h|X)) = \mathrm{KL}(P_{\text{data}}(X)q_\phi(h|X)|p_\theta(h,X)).$$

$$
\begin{aligned}
\frac{\partial}{\partial\theta}\mathrm{KL}(P_{\text{data}}(X)|p_\theta(X)) &= \frac{\partial}{\partial\theta}\mathrm{E}_{P_{\text{data}}(X)}(-\log\sum_h p_\theta(h,X)) - \frac{\partial}{\partial\theta}entropy(P_{\text{data}}(X)) \\
&= \frac{\partial}{\partial\theta}\mathrm{E}_{P_{\text{data}}(X)}(-\log\sum_h p_\theta(h,X)) \\
&= -\mathrm{E}_{P_{\text{data}}(X)}\sum_h \frac{p(h,X)}{\sum_{h'}p(h',X)}\frac{1}{p_\theta(h,X)}\frac{\partial}{\partial\theta}p_\theta(h,X) \\
&= -\mathrm{E}_{P_{\text{data}}(X)}\sum_h p(h|X)\frac{\partial}{\partial\theta}\log p_\theta(h,X) \\
&= -\mathrm{E}_{P_{\text{data}}(X)}\mathrm{E}_{q_\phi(h|X)}\frac{\partial}{\partial\theta}(\log p_\theta(X|h)+\log p(h)) \qquad \text{where} p(h|X)\leftarrow q_\phi(h|X) \\
&= -\mathrm{E}_{P_{\text{data}}(X)}\mathrm{E}_{q_\phi(h|X)}\frac{\partial}{\partial\theta}\log p_\theta(X|h) \\
&= \frac{\partial}{\partial\theta}\mathrm{E}_{P_{\text{data}}(X)}[-\mathrm{E}_{q_\phi(h|X)}\log p_\theta(X|h)]
\end{aligned}
$$

Let q_ϕ be $P_{\text{data}}(X)q_\phi(h|X)$ (where we overload the notation q_ϕ), and let p_θ be $p_\theta(h,X) = p(h)p_\theta(X|h)$ (where we also overload the notation p_θ). Let $Q = \{q_\phi, \forall\phi\}$ and let $P = \{p_\theta, \forall\theta\}$. The VAE problem is to $\min_{p\in P, q\in Q}\mathrm{KL}(q|p)$. Starting from p_0, the VAE iterates the following two steps:

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

(2) $p_{t+1} = \arg\min_{p\in P}\mathrm{KL}(q_{t+1}|p)$.

This is alternating projection. 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 $q_{t+1} = \arg\min_{q\in Q}\mathrm{KL}(p_t|q)$, where we generate dream data from p_t, and learn $q_\phi(h|X)$ from the dream data.

4.5 ELBO

The VAE objective is

$$
\begin{aligned}
&\mathrm{KL}(P_{\text{data}}(X)|p_\theta(X)) + \mathrm{KL}(q_\phi(h|X)|p_\theta(h|X)) \\
&\qquad = \mathrm{E}_{P_{\text{data}}}[\log P_{\text{data}}(X)] - \mathrm{E}_{P_{\text{data}}}[\log p_\theta(X)] + \mathrm{KL}(q_\phi(h|X)|p_\theta(h|X)).
\end{aligned}
$$

Thus minimizing the left-hand side is equivalent to maximizing

$$
\mathrm{E}_{P_{\text{data}}}[\log p_\theta(X)] - \mathrm{KL}(q_\phi(h|X)|p_\theta(h|X)) = \frac{1}{n}\sum_{i=1}^n \left[\log p_\theta(X_i) - \mathrm{KL}(q_\phi(h_i|X_i)|p_\theta(h_i|X_i))\right]
$$

which is a lower bound of the log-likelihood, or evidence lower bound (ELBO).

References

[1] S. Antol, A. Agrawal, J. Lu, M. Mitchell, D. Batra, C. Lawrence Zitnick, and D. Parikh. Vqa: Visual question answering. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2425–2433, 2015.

[2] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[3] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[4] C. Doersch. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*, 2016.

[5] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[6] K. Fukushima and S. Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.

[7] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[8] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[9] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[10] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[11] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[12] D. P. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[14] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[15] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[16] A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

[17] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[18] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[19] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[20] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[21] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[22] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, et al. Going deeper with convolutions. Cvpr, 2015.

[23] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[24] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[25] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, pages 2048–2057, 2015.