Deep Learning Survey

Jicheng Lu

1 Fundamentals in deep learning

1.1 Basic terms in deep learning

1. **Neural network:**
Neural network is the basic structure of deep learning. It is made of numerous interconnected artificial neurons, which pass data between themselves. These neurons are treated linearly with weights and bias, which are updated in the network training process, and then transformed through activation functions to outputs.

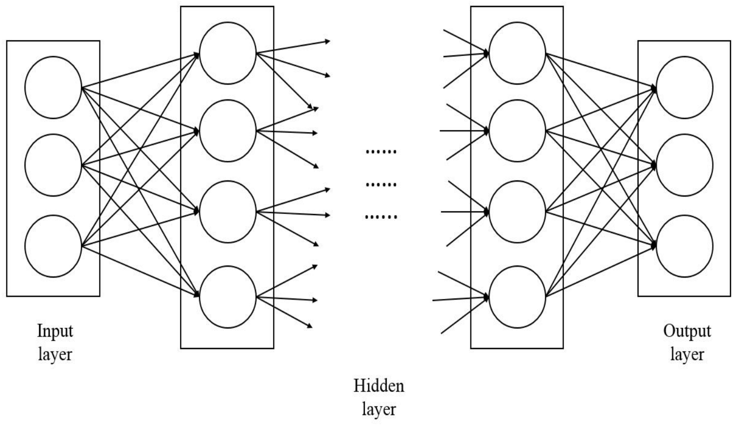


Figure 1: Neural network.

2. **Neuron:**
Neuron is a basic structure of a neural network, just like a neuron in the human brain. A neuron can be seen as a basic processor in the neural network. It receives information (or input) from the upper layer, process the information and output to the next layer. In addition, although a neuron looks like a neuron in the brain, deep learning has nothing to do with biology. Instead, it is a mathematical framework for learning representations from data.

3. **Layers:**
Layers are the elements in neural networks. They can be divided into three categories: input layer, output layer and hidden layer (Fig. 1).

Input layer receives the original data assigned by human, such as image and digits, while output layer presents the final results, such as image recognition outcomes. Both the input layer and output layer are visible to us.

Hidden layer are the ones which perform specific tasks on the incoming data and pass the output generated to the next layer. These hidden layers are automatically trained during the training process.

4. **Weights:**
Weights are usually treated as the parameters of a layer. When inputs enter the neuron, they are multiplied by weights. For example, output = relu(dot(weights, input) + bias).

Initially, the weights are set as random numbers, then they are updated during the model training process. Different weight values represent different importance of features at each layers. One task of deep learning is to find a set of values for the weights of all layers, which is also called learning process. Unlike traditional machine learning technique, deep learning puts emphasis on learning successive layers of increasingly meaningful representations of data.

One advantage of deep learning over traditional machine learning lies in completely automates the feature engineering and simplifies the machine learning workflows. In addition, all the weights and intermediate incremental representations are learned jointly instead of updating them layer by layer.

5. **Bias:**
Bias is added to the weight multiplication (e.g. output = relu(dot(weights, input) + bias)). It is added to change the range of the weight multiplied input. Bias are also parameters of layers in deep learning, which will be learned with weights jointly for all layers.

6. **Activation function:**
The activation function is applied after the inputs are multiplied to weights and added to bias. The activation function transforms the linear production of inputs to the outputs, which will be treated as inputs in the next layers. Several common activation functions are sigmoid, ReLU and softmax.

(a) Sigmoid function:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{1}$$

Sigmoid function gives a smooth range of values between 0 and 1. We can observe the changes in the output with slight changes in the input values.

(b) ReLU function:

$$f(x) = max(x, 0) \tag{2}$$

The benefit of using this function is that it has a constant derivative for all inputs greater than zero, which helps the network to train faster.

(c) Softmax:

The softmax function is similar to sigmoid function. It is usually used in the output layer for multi-class classification problems. The softmax function makes it easy to assign values to each class which can be interpreted as probabilities.

7. **Loss function:**
Loss function is used to measure difference between the predicted value and actual value. The loss function is used in the training process in order to increase the prediction accuracy when we adjust the layer parameters (i.e. weights and bias).

8. **Multiple layer perceptron (MLP):**
MLP is an implementation of several interconnected layers of neurons, forming a feedforward neural network.

9. **Forward propagation:**
Forward propagation refers to the movement of the input through the hidden layers to output layers. In forward propagation, the information travels in one single direction, namely forward. The data only travels forward layers by layers and there is no backward movement.

10. **Gradient descent:**
Gradient descent is an optimization technique for minimizing the difference between the predicted value and actual value, namely minimizing the loss function. We usually take steps proportional to the negative of the gradient of the function:

$$W_{s+1} = W_s - step * gradient(f)(W_s) \tag{3}$$

In order to increase the efficiency, several stochastic gradient descent (SGD) techniques have been developed: mini-batch SGD, ture SGD, and batch SGD. The difference between these SGD techniques lies in the amount of data used at each iteration. Moreover, for the purpose of finding the global minimum instead of local minimum, we use momentum when updating weights based on the previous and current gradients.

11. **Learning rate:**
Learning rate is defined as the amount minimization in the loss function at each iteration. Concretely, the learning rate is the step we take when using gradient descent technique (Eqn. 3). We need to take suitable step value at each iteration in order to find global minimum point with limited time.

12. **Backpropagation:**
When we compare the predicted output with the desired output, we compute the gradient of the loss function with respect to each weight of the neural network. Backpropagation means that we send the gradient descent information back to the neural network to adjust the weights and bias of each layer at each iteration until we find a set of weights that achieves the minimum error between the prediction and true outputs.

13. **Batches:**
When training the neural network, we divide the input data into several batches of equal size randomly and then send the data batches into the neural network. Training the data on batches let the model more generalized, compared to the model trained with entire data.

14. **Epochs:**
Epoch is defined as a single training iteration of all batches in both forward and backpropagation. One epoch is a single forward and backward pass of the entire input data.

15. **Dropout:**
Dropout is a regularization technique which prevents over-fitting of the network. During training, a certain number of neurons in the hidden layer is randomly dropped. This means that the training happens on several architectures of the neural network on different combinations of the neurons.

16. **Batch normalization:**
Batch normalization is to ensure that distribution of data is the same as the next layer hoped to get. While training the neural network, the weights are changed after each step of gradient descent. This changes the shape of data that is sent to the next layer. However, the next layer is expecting the data distribution similar to what it had seen previously. Thus, we use batch normalization before sending the data to the next layer.

17. **Convolutional neural network (CNN):**
CNN is usually applied in computer vision and image recognition. The idea is that we convolve the images to reduce the number of parameters using filter, which is updated like weights and bias during backpropagation for minimizing the loss function. As we slide the filter over the width and height of the input volume, we can produce a 2D activation map that gives the output of that filter at every position. Then we stack these activation maps along the depth dimension and produce the output volume.

18. **Pooling:**
Pooling is common to periodically introduce pooling layers in between the convolution layers. This procedure is intended to reduce the number of parameters and prevent over-fitting. Several common type of pooling is maximum pooling, average pooling, etc.

19. **Padding:**

After applying the filters, the size of convolved layers is reduced. Padding is used to add extra layers of zeros across the images so that the output image has the same size as the input.

20. **Data augmentation:**

Data augmentation refers to the addition of new data derived from the given data, which might be beneficial for prediction. For example, we can rotate the digit to increase the accuracy of the trained model.

21. **Recurrent neural network (RNN):**

RNN is the one in which the output of each neuron is sent back to it for t time stamps. So the connections of recurrent neural networks form a directed cycle (Fig. 2). The cycle within the hidden neuron gives them the capability to store information about the previous words to be able to predict the output. Then the output of the hidden layer is sent again to the hidden layer for t time stamps. Once completing all the time stamps, the output of the layer goes to the next layer.

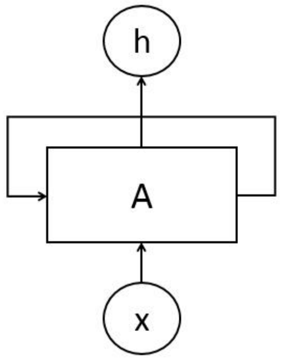


Figure 2: RNN neuron.

22. **Vanishing gradient problem:**

Vanishing gradient problem arises in cases where the gradient of the activation function is very small. During back propagation, when the weights are multiplied with these low gradients, they will become very small and vanish as they go further deep in the network. This makes the neural network to forget the long range dependency. One way to solve this issue is using the activation functions like ReLU which do not have small gradients.

23. **Exploding gradient problem:**

Exploding gradient problem is the opposite of the vanishing gradient problem where the gradient of the activation is too large. This can be solved by clipping the gradient so that it doesn't exceed a certain value.

24. **Long short term memory network (LSTM):**

LSTM is a recurrent neural network which is optimized for learning from and acting upon time-related data.

2 Programming in deep learning

2.1 Existing tools for deep learning

1. **Keras:**

Keras is an open source neural network library in Python. It provides high-level building blocks for developing deep learning models. Keras relies on a specialized, well-optimized tensor library, serving as the backend engine of Keras. There exist three backend implementations: TensorFlow, Theano, and CNTK (Microsoft Cognitive Toolkit).

2. **TensorFlow:**
TensorFlow is a Python library, developed by Google, used to implement deep learning networks. It perform the computation using dataflow graph.

3. **Theano:**
Theano is a Python library that allows us to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. It features in tight integration with NumPy, transparent use of a GPU, efficient symbolic differentiation, speed and stability optimizations, dynamic C code generation and extensive unit-testing and self-verification.

4. **CNTK:**
CNTK is a deep learning framework developed by Microsoft. It describes neural networks as a series of computational steps via a directed graph.

3 Deep learning application

3.1 Written digit recognition

1. **Data:**
MNIST is a set of computer vision dataset, which consists of many images of handwritten digits. For each digit, it also contains a label such that it can tell us what the digit is. The MNIST dataset can be obtained from TensorFlow tutorial.

2. **One-layer Neural Network:**
In this section, we developed a one-layer neural network Python code to recognize the written digits from the MNIST dataset.

We first obtain the MNIST data from the TensorFlow tutorial, which consists of $55,000$ training images, $10,000$ test images, and $5,000$ validation images. Note that each image is 28 pixels by 28 pixels. Thus, there are 28x28 = 784 numbers representing each image.

After importing the dataset, we start to define the model parameters, such as the weight(W) and bias (b). Note that the weight is a 784x10 matrix, since there are 10 possible outcomes for the digit recognition (0$\sim$9). In this case, we use the classic softmax function (Eqn. 4) to convert the image inputs to probabilities that can tell us the classification results.

$$y(x) = softmax(W \cdot x + b) \tag{4}$$

Before we start to train the model, we need to define a loss function and select an appropriate optimizer. In this case, the cross entropy are selected as the loss which is to measure how inefficient our predictions are for describing the truth. The cross entropy is given as follows:

$$L(y) = -\sum_i y_i^{'} \cdot log(y_i) \tag{5}$$

where y is the prediction and y' is the truth.

Given the loss function, we use the Gradient Descent Algorithm in TensorFlow with a learning rate of 0.7 to minimize the cross entropy. Then we train the model with $5,000$ steps using stochastic training technique, which means we select small batches of random data from the dataset for training in each step. The Python statements of this one-layer neural network are given in Box 3.1.1.

Box 3.1.1 Python statement of one-layer neural network for the digit recognition.

```
import tensorflow as tf

## get the data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data", one_hot=True)

## define the model parameters
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))

y = tf.nn.softmax(tf.matmul(x, W) + b)

## define the loss function and optimizer
y_true = tf.placeholder(tf.float32, [None, 10])

cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(labels=y_true, logits=y))
# cross entropy using unnormalized logits

learning_rate = 0.7
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(cross_entropy) # use gradient
descent optimizer

correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_true,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32)) # calculate the accuracy of prediction

## initialization
sess = tf.InteractiveSession()
tf.global_variables_initializer().run()

## train the model
training_iters = 5000
for i in range(training_iters):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    if i%100 == 0:
      train_accuracy = accuracy.eval(feed_dict=x: mnist.train.images, y_true: mnist.train.labels)
      print("step %d, training accuracy %g"%(i, train_accuracy))
    sess.run(train_step, feed_dict=x:batch_xs, y_true:batch_ys)

## train and test performance
print("Accuracy on training data = ", sess.run(accuracy,
         feed_dict=x:mnist.train.images, y_true: mnist.train.labels))
print("Accuracy on test data = ", sess.run(accuracy,
         feed_dict=x:mnist.test.images, y_true: mnist.test.labels))
```

3. Multilayer Convolutional Neural Network:

In order to improve the performance, we consider upgrading the neural network with a multilayer convolutional neural network. As usual, we start from defining the model parameters. Here we initialize the weights with truncated normal distribution. This strategy can overcome the saturation of functions (e.g. sigmoid), where the neurons stop learning if the values is too big or small. Moreover, we set the initial bias a little bigger than zero in order to avoid dead neuron.

In this case, we develop two convolutional layers and one fully connected layer with maximum pooling and padding techniques. We also apply the dropout method before the final output layer to reduce over-fitting.

The Python statements of the multilayer convolutional neural network are given in Box 3.1.2.

Box 3.1.2 Python statement of multilayer CNN for the digit recognition.

```
import tensorflow as tf

## get the data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data", one_hot=True)

def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)

def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                strides=[1, 2, 2, 1], padding='SAME')

x = tf.placeholder(tf.float32, [None, 784])

## first convolutional layer
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])

x_image = tf.reshape(x, [-1,28,28,1])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

## second convolution layer
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)

## fully connected layer
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

## dropout
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

Box 3.1.2 Python statement of multilayer CNN for the digit recognition (Continued).

```
## output layer
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2

## define loss function and optimizer
y_true = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_true, logits=y_conv))

learn_rate = 1e-4
train_step = tf.train.AdamOptimizer(learn_rate).minimize(cross_entropy)

correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_true,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

## initialization
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())

## model training
train_iters = 10000
for i in range(train_iters):
    batch = mnist.train.next_batch(50)
    if i%100 == 0:
      train_accuracy = accuracy.eval(feed_dict=x: batch[0], y_true: batch[1], keep_prob: 1.0)
      print("step %d, training accuracy %g"%(i, train_accuracy))
    train_step.run(feed_dict=x: batch[0], y_true: batch[1], keep_prob: 0.5)

print("test accuracy %g"%accuracy.eval(feed_dict=x: mnist.test.images, y_true: mnist.test.labels, keep_prob:
1.0))
```

4.**Performance Comparison between the two Neural Networks:**

The training and testing performance is presented in Table. 1. We can observe that the multilayer convolutional neural network is much better than the one-layer neural network, because of its deeper network and more complex optimizer. However, the accuracy can still be improved in the future with more advanced techniques.

Table 1: Performance comparison between different neural networks

	One-layer NN	multilayer CNN	Improvement (%)
Training accuracy (%)	92.27	98.00	6.21
Testing accuracy (%)	92.33	99.03	7.26

4 Recurrent Neural Networks (RNN)

4.1 Introduction about RNN

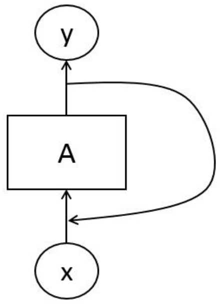
Recurrent Neural Network (RNN) is a class of neural networks that can predict the future and exploit the sequential nature of their inputs. Such inputs can be text, speech, stock price, time series and anything else where the occurrence of an element in the sequence is dependent on the elements that appeared before it.

An RNN can be thought of as a graph of RNN cells, where each cell performs the same operation on every element in the sequence. RNNs are very flexible and have been used to solve problems such as stock price prediction, natural language processing, sentiment analysis, image captioning, machine translation, etc.

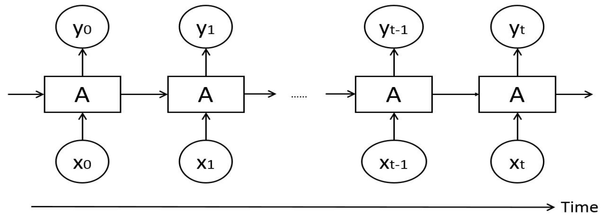
4.2 Simple RNN cells

4.2.1 Recurrent neurons

Traditionally, the activations flow only in one direction, from input layer to output layer, which means all the inputs are independent with each other. However, when it comes to sequence data, this situation has changed: the output from previous time step also affects the input in the current time step. This is actually how an RNN works. An RNN has not only a feedforward neural network, but also connections pointing backward. Fig. 3(a) shows the simplest recurrent neuron, receiving inputs, producing an output and sending the output back to itself. We can also unroll the network through time, as is shown in Fig. 3(b).



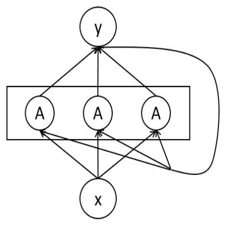
(a) A recurrent neuron.



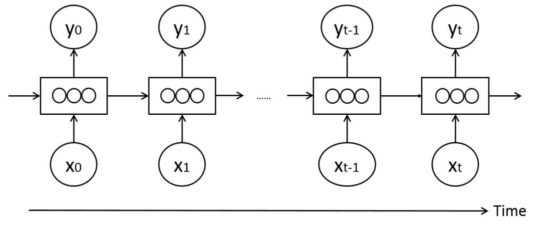
(b) A recurrent neuron unrolled through time.

Figure 3: One recurrent neuron.

Then, we can easily build a layer of recurrent neurons. At each time step t, every neuron receives both the input vector x_t and the output vector from the previous time step y_{t-1} (shown in Fig.4(a)). Fig.4(b) unrolls the layer of recurrent neurons through time.



(a) A layer of recurrent neurons.



(b) A layer of recurrent neurons unrolled through time.

Figure 4: One layer of recurrent neurons.

As we can see above, each recurrent neuron should have two sets of weights: one for the inputs (x_t) and the other for the outputs of the previous time step (y_{t-1}). Suppose the weights for the inputs and outputs are W_x and W_y ,respectively. We can compute the output a recurrent layer's output just like before for a whole mini-batch (Eq. 6):

$$Y_t = \phi(X_t \cdot W_x + Y_{t-1} \cdot W_y + b) = \phi([X_t, Y_{t-1}] \cdot [W_x, W_y]^T + b) \tag{6}$$

where Y_t is and $m \times n_{neurons}$ matrix containing the layer's outputs at time step t for m instances in the mini-batch (m is the number of instances in the mini-batch and $n_{neurons}$ is the number of neurons). X_t is and $m \times n_{inputs}$ matrix containing the inputs for m instances (n_{inputs} is the number of inputs). W_x is and $n_{inputs} \times n_{neurons}$ matrix containing the connection weights for the inputs of the current time step. W_y is and $n_{neurons} \times n_{neurons}$ matrix containing the connection weights for the outputs of the previous time step. b is a vector of size $n_{neurons}$ containing each neuron's bias term.

4.2.2 Memory cells

From the definition of RNN, we know that the output of a recurrent neuron at time step t is a function of all the inputs from previous time steps, which means it has memory.

In general, at each time step t, we send the hidden state (h_t) back to the inputs for each recurrent neuron. The hidden state (h_t) is also a function of some inputs at that time step and its state at the previous time step: $h_t = f(h_{t-1}, x_t)$.

4.3 RNN topologies

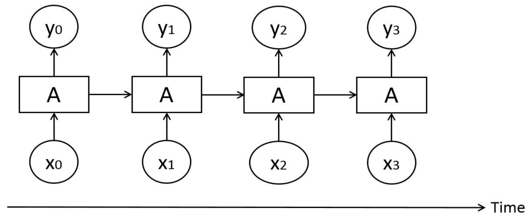
Generally, there are four types of RNN topologies: sequence-to-sequence, sequence-to-vector, vector-to-sequence, and delayed sequence-to sequence. All these topologies derive from the same basic structure shown in Fig.3(b).

First, an RNN can take a sequence of inputs and produce a sequence of outputs (Fig. 5(a)). This type of network is useful for predicting time series such as stock prices: input the prices over the last N days, and it will output the prices shifted by one day into the future.

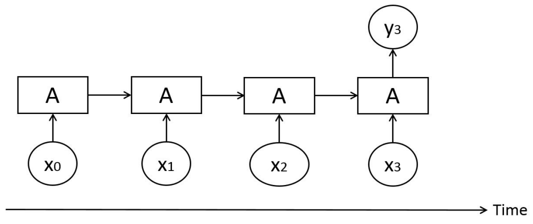
Secondly, the network can be fed a sequence of inputs, and ignore all outputs except for the last one. (Fig. 5(b)). For example, we can use the sequence-to-vector structure in the sentiment analysis: feed a sequence of words about a movie review, and output a single sentiment score.

Conversely, we can also feed the network a single input at the first time step, and let it output a sequence (Fig. 5(c)). For example, we can use this vector-to-sequence network for image captioning: input an image and output a caption for that image.

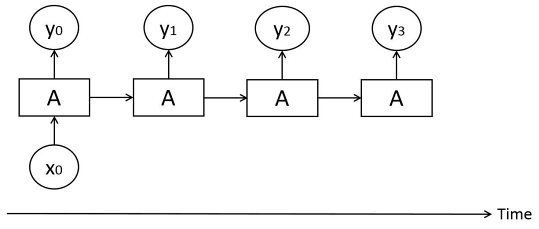
Lastly, we can combine a sequence-to-vector network (”encoder”) with a vector-to-sequence network (”decoder”), as is shown in Fig. 5(d). For example, we can use this type of network to translate one language to another. We feed the network a sentence in one language, the encoder converts it into a single vector, and then the decoder converts this vector into a sentence in another language. This encoder-decoder model works much better than the single sequence-to-sequence RNN, because the last words of a sentence can affect the first words of the translation, so we need to hear the whole sentence before starting translation.



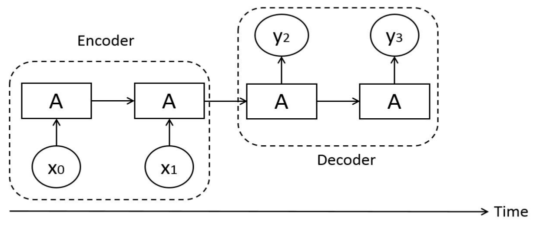
(a) Sequence-to-sequence.



(b) Sequence-to-vector.



(c) Vector-to-sequence.



(d) Delayed sequence-to sequence.

Figure 5: RNN topologies.

4.4 Training RNNs

To train an RNN, we first need to unroll it through time, and then use the regular backpropagation. This strategy is called backpropagation through time (BPTT).

Fig. 6 presents a typical RNN training process. First, there is a regular forward flow passing through the unrolled network, which is represented by the solid arrows. Then, a cost function is evaluated using part of the outputs, $Cost(y_2, y_3)$, and the gradients of this cost function are propagated backward through the unrolled network, which is represented by the dashed arrows. Finally, the model parameters are updated using the gradients obtained during BPTT. Note that the backward flow only goes through the outputs used in the cost function instead of all the outputs. Moreover, we need apply dropout technique to avoid over-fitting when training the model.

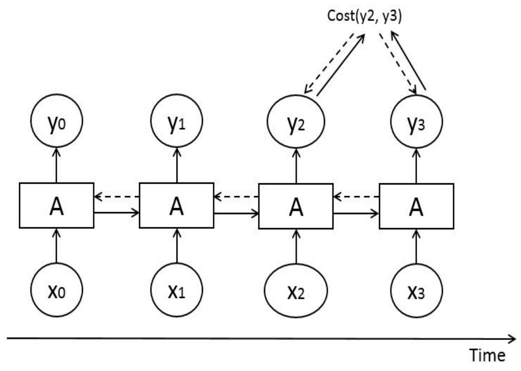


Figure 6: Backpropagation through time.

4.5 Deep RNNs

From the RNN knowledge above, we can stack multiple layers of RNN cells and get a deep RNN. Fig.7 presents a typical unrolled RNN.

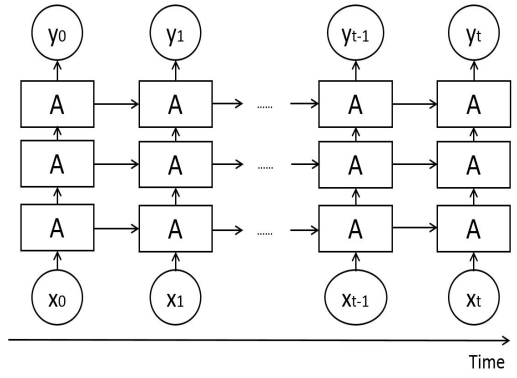


Figure 7: Deep RNN.

4.6 Difficulties in training RNNs

When we train an RNN on long sequences, we need to build a very deep network and run it over many time steps. Like any deep neural network, it may suffer from the vanishing/exploding gradients problem and take forever to train.

The vanishing/exploding gradients problem can be briefly explained as follows: consider the case where the individual gradients of a hidden state with respect to the previous one is less than one (i.e. $\frac{\partial h_j}{\partial h_{j-1}} < 1$).

As we backpropagate across multiple time steps, the product of gradients get smaller and smaller, leading to the vanishing gradients problem. On the other hand, if the gradients are larger than one (i.e. $\frac{\partial h_j}{\partial h_{j-1}} > 1$), the product of gradients get bigger and bigger, leading to the exploding gradients problem.

There are several methods used to alleviate the these problems, such as proper initialization of parameters, non-saturating activation functions, batch normalization, gradient clipping and faster optimizers. However, they still cannot improve the training speed, when RNN deals with long sequences.

One of the simplest solution to these problems is using truncated backpropagation through time: unroll RNN over a limited number of time steps during training. It reduces the sequences and includes both old and recent data. However, this technique doesn't have long-term memory, and may have some problems when we deal with the topics where the long-term data is very important.

Besides the long training time, the RNN is also facing the problem of long-term memory: the memory of the first inputs gradually fades away during training. This is where the LSTM and GRU cells come in to solve this problem.

4.7 LSTM cells

The long short-term memory (LSTM) cell is a variant of RNN that is able to detect long-term dependencies in the data and train faster.

LSTMs implement recurrence with four fully-connected layers interacting in a specific way. There are also two states in LSTM cell: short-term state (h_t) and long-term state (c_t). Fig. 8 presents a typical LSTM cell.

As we can see from the LSTM cell, there are three gates in the LSTM cell: forget gate, input gate, and output gate. The function of each gate is given as follows:

a) Forget gate: erase part of the long-term state;

b) Input gate: add part of g_t to the long-term state;

c) Output gate: read part of long-term state and output to h_t and y_t.

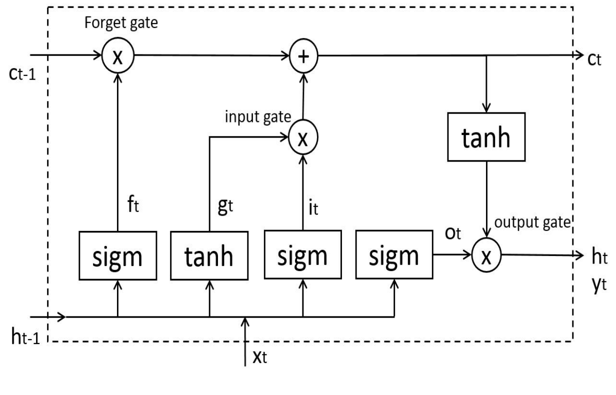


Figure 8: LSTM cell.

Eq. 7 summarizes how to compute the cell's long-term state, its short-term state, and its output at each time step for a single instance.

$$\begin{aligned}
i_t &= \sigma(W_{xi}^T \cdot x_t + W_{hi}^T \cdot h_{t-1} + b_i), \\
f_t &= \sigma(W_{xf}^T \cdot x_t + W_{hf}^T \cdot h_{t-1} + b_f), \\
o_t &= \sigma(W_{xo}^T \cdot x_t + W_{ho}^T \cdot h_{t-1} + b_o), \\
g_t &= tanh(W_{xg}^T \cdot x_t + W_{hg}^T \cdot h_{t-1} + b_g), \\
c_t &= f_t \otimes c_{t-1} + i_t \otimes g_t, \\
y_t &= h_t = o_t \otimes tanh(c_t).
\end{aligned} \tag{7}$$

where $W_{xi}, W_{xf}, W_{xo}, W_{xg}$ are the weight matrices of each of the four layers for their connection to the input vector x_t. $W_{hi}, W_{hf}, W_{ho}, W_{hg}$ are the weight matrices of each of the four layers for their connection to the previous short-term state h_{t-1}. b_i, b_f, b_o, b_g are the bias terms for each of the four layers.

In short, the LSTM cell can learn to recognize an important input, store it in the long-term state, and learn to preserve it as long as it is needed, and learn to extract it whenever it is needed. This is why the LSTM cell is becoming popular.

4.8 GRU cells

The Gated Recurrent Unit (GRU) cell is a simplified version of LSTM cell. Fig. 9 gives a typical GRU cell.

Comparing to the LSTM cell, the GRU cell merges both state vectors into a single vector h_t. Moreover, instead of the forget, input, output gates in LSTM cell, the GRU cell only has two gates: update gate z, and reset gate r. The update gate defines how much previous memory to preserve and the reset gate defines how to combine the new input with the previous memory.

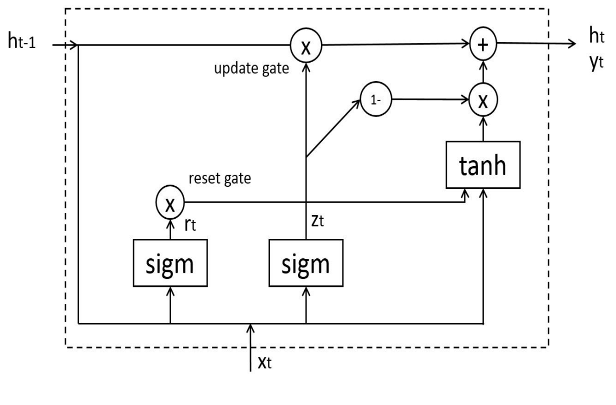


Figure 9: GRU cell.

Eq. 8 summarizes how to compute the cell's state at each time step for a single instance.

$$\begin{aligned}
z_t &= \sigma(W_{xz}^T \cdot x_t + W_{hz}^T \cdot h_{t-1} + b_z), \\
r_t &= \sigma(W_{xr}^T \cdot x_t + W_{hr}^T \cdot h_{t-1} + b_r), \\
g_t &= tanh(W_{xg}^T \cdot x_t + W_{hg}^T \cdot (r_t \otimes h_{t-1}) + b_g), \\
y_t &= h_t = z_t \otimes h_{t-1} + (1 - z_t) \otimes g_t.
\end{aligned} \tag{8}$$

where W_{xz}, W_{xr}, W_{xg} are the weight matrices of each of the three layers for their connection to the input vector x_t. W_{hz}, W_{hr}, W_{hg} are the weight matrices of each of the three layers for their connection to the previous short-term state h_{t-1}. b_z, b_r, b_g are the bias terms for each of the three layers.

Both GRU and LSTM have comparable performance and the selection of these two cells are dependent on a specific task. While GRUs are faster to train and need less data to generalize, an LSTM's expressive poser may lead to better results. Moreover, both LSTMs and GRUs are drop-in replacement for the simple RNN cell. Thus, there should not be any side effects but better performance.

4.9 Bidirectional RNNs

In some situation, it is possible that the output is dependent on not only the previous outputs but also the future outputs. This is especially true in Natural Language Processing (NLP), where the attributes of word or phrase we are trying to predict may depend on the context given by the entire enclosing sentence, not just the words that came before it. This is where bidirectional RNNs come in. The bidirectional RNNs place equal emphasis on the beginning and end of the sequence, and increase the data available for training.

Bidirectional RNNs are two RNNs stacked on top of each other, reading the input in opposite directions. The output at each time step will be based on the hidden state of both RNNs.

4.10 Other RNN variants

There are also some other variants of the RNN cell.

One popular LSTM variant is adding peephole connections, which means that the gate layers are allowed to peek at the cell state. Come back to the Fig. 8, we add the previous long-term state c_{t-1} as an input to the controllers of forget gate and input gate, while adding current long-term state c_t as an input to the controller of the output gate.

Another LSTM variant, that ultimately led to the GRU, is to use coupled forget and output gates. Decisions about what information to forget and what to acquire are made together, and the new information replaces the forgotten information.

5 Atuoencoders

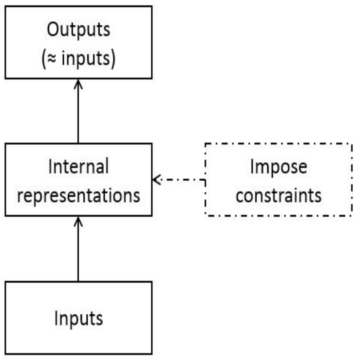
5.1 Introduction about autoencoders

Autoencoders are artificial neural networks that are capable of learning efficient representations of the input data without any supervision. The useful data representations in autoencoders are called codings. These codings usually have lower dimension than the input data, making autoencoders useful for dimensionality reduction. The primary task of autoencoders is to learn to duplicate the inputs to outputs. At fist glance, it may be seen as unnecessary, but it will become more difficult if we impose constraints to the network, such as adding noise to inputs, limiting the size of the internal representation, or developing latent space. Besides, the autoencoders can also be trained for generating new data which looks very similar to the training data. For example, we can train an autoencoder using many human face pictures, and when we input a new face, it can generate a new face.

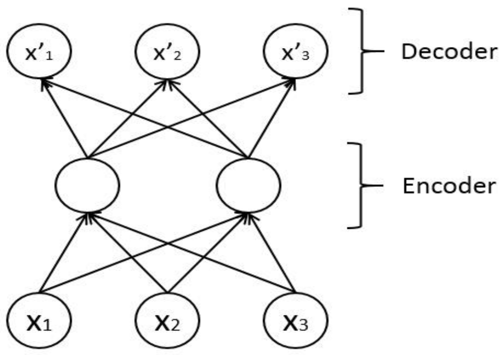
In short, autoencoders can be used for dimension reduction, feature detector, unsupervised pretraining of neural network, and generative models. The key idea of autoencoders lies in finding a useful data feature (or pattern) in the input data and recovering the original data. Currently, autoencoders have been applied in many areas, such as music generation, dialogue generation, image generation, speech synthesis, molecule design, etc.

5.2 Stacked autoencoders

A typical autoencoder shares the same structure as the multi-layer perception (MLP). It consists of two parts: an encoder (converting the inputs to the internal representations) and a decoder (converting the internal representations to outputs). The typical flow chart of the autoencoder is presented in Fig.10(a) and a simple autoencoder is given in Fig.10(b). Note that the number of neurons in the output layer of an autoencoder must be equal to the number of inputs in order to recover the original data.



(a) Flow chart of a typical encoder.



(b) A simple autoencoder.

Figure 10: Typical autoencoder structure.

Then, we come to the stacked autoencoders. Like other neural networks, autoencoders can also have multiple hidden layers. These autoencoders are so-called stacked autoendcoders (or deep autoencoders). Adding more layers helps the autoencoder learn more complex data features (codings). A typical stacked autoencoder is presented in Fig.11. We can observe that a stacked autoencoder is symmetrical with respect to the central hidden layer.

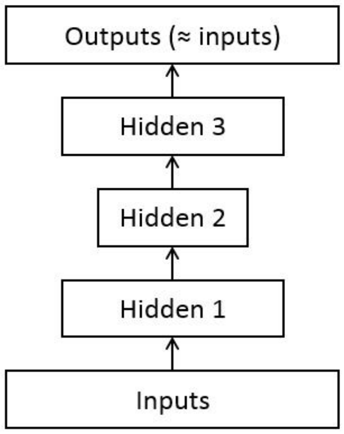


Figure 11: Stacked autoencoder.

5.3 Training autoencoders

Before starting to train the autoencoder, we need to tie the weights of the decoder layers to the weights of the encoder layers, due to the symmetry of its architecture. This can help reduce the total number of weights, speed up the training process and limit the risk of over-fitting. Note that the bias are never tied during training.

Generally, there are two training strategies. The first one is multiphase training, where we train one shallow autoencoder at a time and then stack all of them into a single stacked autoencoder. Fig.12 shows a typical multiphase training process.

During the first phase of training, the first autoencoder learns to reconstruct the inputs with the first hidden layer. In the second phase, on the other hand, the second autoencoder learns to reconstruct the output of the first hidden layer. After training all the parameters in those hidden layers, we stack all of them and obtain an entire autoencoder. Note that in phase II, the parameters in the first hidden layer are fixed and we only

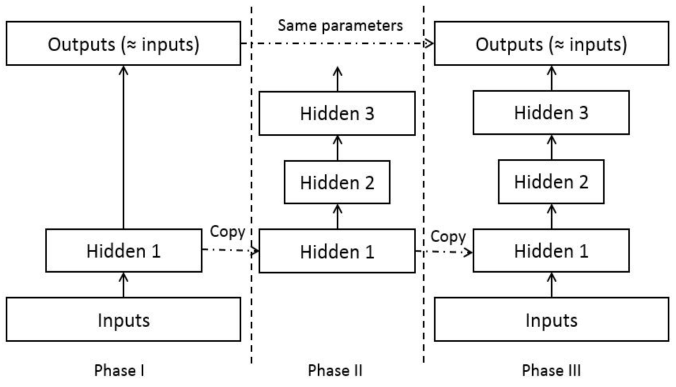


Figure 12: Stacked autoencoder.

train for the parameters in "Hidden 2" and "Hidden 3". This strategy can let us build any deeper stacked autoencoders.

Another training strategy is to train these networks in totality with better activation and regularization functions.

5.4 Unsupervised pretraining using stacked autoencoders

As we have discussed in Section 5.1, one of the functions of autoencoders is to unsupervised pretraining of neural networks. This can be achieved as follows. We first train a stacked autoencoder using all the data without labels, and then reuse the lower layers to create a neural network for a specific task. Finally, we train the whole network using the labeled data. (Fig.13) This operation can help us train a high-performance model using only little training data, because we only use the low-level features in the input data.

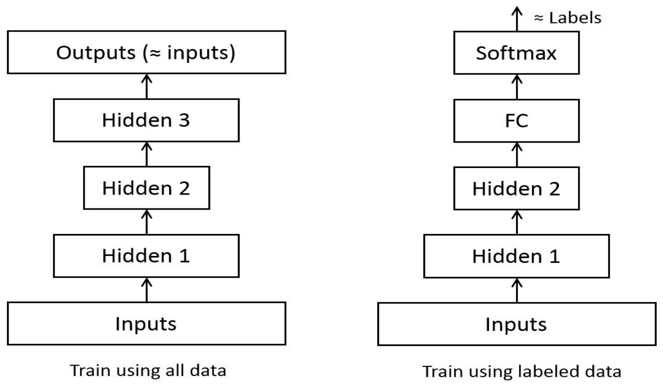


Figure 13: Unsupervised pretraining using stacked autoencoders.

5.5 Variational autoencoders (VAEs)

The key idea of the variational autoencoders is to develop a low-dimensional latent space of representation (e.g., vector space) where any point can be mapped to the original inputs. Fig.14 shows how the variational autoencoder work. Instead of directly producing a coding for a given input, the encoder turns the local outputs into the parameters of a Gaussian distribution with mean μ and standard deviation σ. This means that the latent space is based on the Gaussian distribution. Once the latent space is generated, we can sample points from it, either deliberately or randomly, and decode them into the original space. Note that one advantage of using VAEs is that it can learn a well-structured latent space where specific directions encode a meaningful axis of variance in the data. For example, when we are editing an image, there may be a smile vector in the latent space. If we add this vector to another latent vector, which represents a certain face, and map it back to the original space, we may obtain a smiling face in the end.

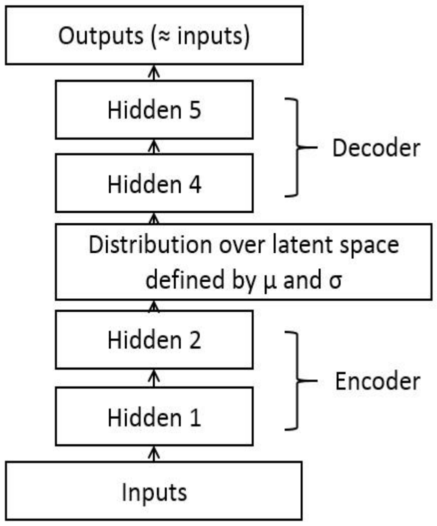


Figure 14: A variational autoencoder.

5.6 Generative Adversarial Networks (GANs)

Another important type of autoencoders is the Generative Adversarial Network (GAN). Like VAEs, GANs are also learning latent spaces for the inputs. A GAN is composed of two parts: generator and discriminator. The generator is to produce the fake data, and the discriminator is to distinguish actual data from this fake data. Thus, during training, the capability of the generator is improving while the discriminator is adapting to the generator's capability, setting a high standard of realism for the generated data (i.e., image). However, unlike VAEs, the latent space, generated by GANs, has little meaningful structure. More specifically, it is not continuous.

5.7 Denoising autoencoders

An alternative to the autoencoders mentioned above is to add noise to its inputs, training it to recover the original, noise-free inputs while finding meaningful patterns from the data. Fig.15 shows a typical denoising autoencoder with either Gaussian noise or dropout.

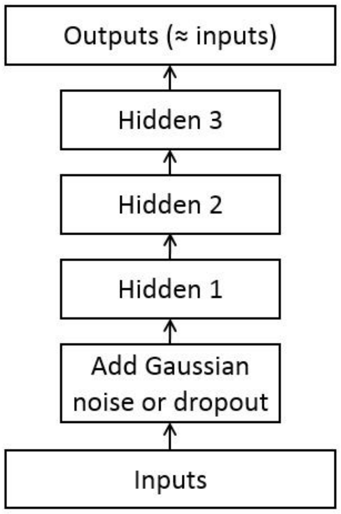


Figure 15: A denoising autoencoder.

5.8 Sparse autoencoders

Another kind of constraint that often leads to good feature extraction is sparsity: by adding a sparsity loss to the reconstruction loss. We first measure the actual sparsity of the coding layer at each training iteration, and then we penalize the neurons that are too active by adding a sparsity loss (e.g., mean square error or KullbackCLeibler divergence). Moreover, in order to control the balance between the sparsity loss and reconstruction loss, we introduce a sparsity weight hyperparameter. Based on these procedure, we just minimize the combined cost function during training as the way we have discussed before.

5.9 Other autoencoders

There are also a few more types of autoencoders. such as contractive autoencoder (CAE), stacked convolutional autoencoder, generative stochastic network (GSN), and winner-take-all autoencoder (WTA).

Contractive autoencoder: Add constraints such that the derivatives of the codings with regards to the inputs are small.

Stacked convolutional autoencoder: Construct the layers in the autoencoder using convolutional technique.

Generative stochastic network: A general version of denoising autoencoders with the additional capability to generate data.

Winner-take-all autoencoder: It is similar to sparse autoencoders. During training, we only preserve the top k% activations for each neuron over the training batch, and set the rest to zero.

5.10 Summary

The characteristics among different types of autoencoders are summarized in Table.2.

Table 2: Characteristics of different autoencoders

Autoencoders	**Characteristics**
Variational autoencoder	• Develop a low-dimensional latent space defined by a mean and a standard deviation; • Outputs are partly determined by chance; • New instance can be generated by sampling from the latent space; • The latent space is well-structured and meaningful.
Generative adversarial network	• Generator and discriminator compete with each other; • Lead to increasingly realistic fake data and robust codings; • The latent space is not well-structured or continuous.
Denoising autoencoder	• Add noise or dropout to inputs.
Sparse autoencoder	• Add a sparsity loss to penalize the neurons that are too active; • Introduce a sparse weight hyperparameter to control the balance between reconstruction loss and sparsity loss.
Contractive autoencoder	• Constrain the derivatives of the codings with regards to the inputs to be small.

Stacked convolutional autoencoder	• Construct the autoencoder using convolutional layers.
Generative stochastic network	• A general version of denoising autoencoders.
Winner-take-all autoencoder	• Similar to sparse autoencoders; • Preserve the top k% activations for each neuron over the training batch and set the rest to zero.