Chapter 11

Deep Learning for Feature Representation

Suhang Wang

Arizona State University

Huan Liu

Arizona State University

11.1 Introduction

Deep learning methods have become increasingly popular in recent years because of their tremendous success in image classification [19], speech recognition [20] and natural language processing tasks [60]. In fact, deep learning methods have regularly won many recent challenges in these domains [19]. The great success of deep learning mainly comes from specially designed structures

of deep nets, which are able to learn discriminative non-linear features that can facilitate the task at hand. For example, the specially designed convolutional layers of CNN allow it to extract translation-invariant features from images while the max pooling layers of CNN help to reduce the parameters to be learned. In essence, the majority of existing deep learning algorithms can be used as powerful feature learning/extraction tools, i.e., the *latenet features* extracted by deep learning algorithms are the learned new representations. In this chapter, we will review classical and popular deep learning algorithms and explain how they can be used for feature representation learning. We will also discuss how they are used for hierarchical and disentangle representation learning, and how they can be applied for various domains.

11.2 Restricted Boltzmann Machine

A restricted Boltzmann machine (RBM) is an undirected graphical model that defines a probability distribution over a vector of observed, or visible, variables $\mathbf{v} \in \{0,1\}^m$ and a vector of latent, or hidden, variables $\mathbf{h} \in \{0,1\}^d$, where m is the dimension of input features and d is the dimension of the latent features. It is widely used for unsupervised representation learning. For example, $\mathbf{v}$ can be the bag-of-word representation of documents or the vectorized binary images and $\mathbf{h}$ *is the learned representation for the input data*. A typical choice is that $d < m$, i.e., learning compact representation. Figure 11.1(a) gives a toy example of an RBM. In the figure, each node of the hidden layer is connected to each node in the visible layer, while there are no connections between hidden nodes or visible nodes. Figure 11.1(b) is a simplified representation of RBM, where the connection details between hidden layers and visible layers are simplified. We will begin by assuming both $\mathbf{v}$ and $\mathbf{h}$ as binary vectors, i.e., elements of $\mathbf{v}$ and $\mathbf{h}$ can only take the value of 0 or 1. The extension to real valued input $\mathbf{x}$ will be introduced 11.2.2. An RBM defines a joint probability over $\mathbf{v}$ and $\mathbf{h}$ as,

$$P(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} \exp(-E(\mathbf{v}, \mathbf{h})) \tag{11.1}$$

where Z is the partition function defined as $Z = \sum_{\mathbf{v}} \sum_{\mathbf{h}} \exp(-E(\mathbf{v}, \mathbf{h}))$, and E is an energy function given by

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{h}^T \mathbf{W} \mathbf{v} - \mathbf{b}^T \mathbf{h} - \mathbf{c}^T \mathbf{v} \tag{11.2}$$

here $\mathbf{W} \in \mathbb{R}^{d \times m}$ is a matrix of pairwise weights between elements of $\mathbf{v}$ and $\mathbf{h}$ (see figure 11.1(a)), while $\mathbf{b} \in \mathbb{R}^{d \times 1}$ and $\mathbf{c} \in \mathbb{R}^{m \times 1}$ are biases for the hidden and visible variables, respectively[1].

[1]For simplicity, bias terms are not shown in Figure 11.1.

Since there are no explicit connections between hidden units in an RBM, given a randomly selected training data $\mathbf{v}$, the hidden units are independent of each other, which gives $P(\mathbf{h}|\mathbf{v}) = \prod_{i=1}^{d} P(h_i|\mathbf{v})$, and the binary state, h_i, $i = 1, \ldots, d$, is set to 1 with conditional probability given as,

$$P(h_i = 1|\mathbf{v}) = \sigma\big(\sum_{j=1}^{m} W_{ij} v_j + b_i\big) \tag{11.3}$$

where $\sigma(\cdot)$ is the sigmoid function defined as $\sigma(x) = (1 + \exp(-x))^{-1}$. Similarly, given $\mathbf{h}$, the visible units are independent of each other. Thus, we have $P(\mathbf{v}|\mathbf{h}) = \prod_{j=1}^{m} P(v_j|\mathbf{h})$, and the binary state, v_j, $j = 1, \ldots, m$, is set to 1 with conditional probability given as

$$P(v_j = 1|\mathbf{h}) = \sigma(\sum_{i=1}^{d} W_{ij} h_i + v_j) \tag{11.4}$$

With the simple conditional probabilities given by Eq.(11.3) and Eq.(11.4), sampling from $P(\mathbf{h}|\mathbf{v})$ and $P(\mathbf{v}|\mathbf{h})$ becomes very efficient. RBMs have generally been trained using gradient ascent to maximize the log-likelihood $l(\boldsymbol{\theta})$ for some set of training vectors $\mathbf{V} \in \mathbb{R}^{m \times n}$, where $\boldsymbol{\theta} = \{\mathbf{W}, \mathbf{b}, \mathbf{c}\}$ is the set of variables to be optimized. The log-likelihood $l(\boldsymbol{\theta})$ is written as

$$l(\boldsymbol{\theta}) = \frac{1}{n} \log P(\mathbf{V}) = \frac{1}{n} \sum_{i=1}^{n} \log P(\mathbf{v}_i) \tag{11.5}$$

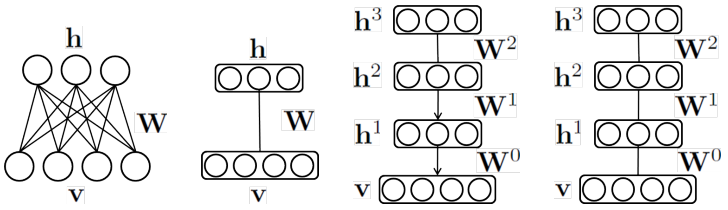
The derivative of $\log P(\mathbf{v})$ w.r.t variable $\mathbf{W}$ is given as,

$$\frac{\partial \log P(\mathbf{v})}{\partial \mathbf{W}} = \sum_{\mathbf{h}} P(\mathbf{h}|\mathbf{v}) \mathbf{h}\mathbf{v}^T - \sum_{\tilde{\mathbf{v}}} \sum_{\mathbf{h}} P(\tilde{\mathbf{v}}, \mathbf{h}) \mathbf{h}\tilde{\mathbf{v}}^T \tag{11.6}$$

where $\tilde{\mathbf{v}} \in \{0, 1\}^m$ is an m-dimensional binary vector. The first term in Eq.(11.6) can be computed exactly. This term is often referred to as the *positive* gradient. It corresponds to the expected gradient of the energy with respect to $P(\mathbf{h}|\mathbf{v})$. The second term in Eqs.(11.6) is known as the *negative* gradients, which is expectation over the model distribution $P(\mathbf{v}, \mathbf{h})$. It is intractable to compute the negative gradients exactly. Thus, we need to approximate the negative gradients by sampling $\mathbf{v}$ from $P(\mathbf{v}|\mathbf{h})$ and sampling $\mathbf{h}$ from $P(\mathbf{h}|\mathbf{v})$ by maintaining a Gibbs chian. For more details, we encourage readers to refere to Contrastive Divergence [62].

11.2.1 Deep Belief Networks and Deep Boltzmann Machine

RBMs can be stacked and trained in a greedy manner to form so-called Deep Belief Networks (DBN) [21] . DBNs are graphical models which learn



(a) RBM (b) RBM (c) 3-layer DBN (d) 3-layer DBM

FIGURE 11.1: An illustration of RBM, DBN and DBM

to extract a deep hierarchical representation of the training data. They model the joint distribution between observed vector $\mathbf{v}$ and the l hidden layers as:

$$P(\mathbf{x}, \mathbf{h}^1, \mathbf{h}^2, \ldots, \mathbf{h}^l) = \left(\prod_{k=0}^{l-2} P(\mathbf{h^k} | \mathbf{h^{k+1}}) \right) P(\mathbf{h}^{l-1}, \mathbf{h}^l) \tag{11.7}$$

where $\mathbf{v} = \mathbf{h}^0$. $P(\mathbf{h}^{k-1}|\mathbf{h}^k)$ is a conditional distribution for the visible units conditioned on the hidden units of the RBM at level k, and $P(\mathbf{h}^{l-1}, \mathbf{h}^l)$ is the visible-hidden joint distribution in the top-level RBM. This is illustrated in Figure 11.1(c). DBN is able to learn hierarchical representation [33]. The low-level hidden representation such as $\mathbf{h}^1$ captures *low-level features* while the high-level hidden representation such as $\mathbf{h}^3$ captures more complex *high-level features*. Training of DBN is done by greedy layer-wise unsupervised training [21]. Specifically, we first train the first layer as an RBM with the raw input $\mathbf{v}$. From the first layer, we obtain the latent representation being the mean activations $P(\mathbf{h}^1|\mathbf{h}^0)$ or samples of $P(\mathbf{h}^1|\mathbf{h}^0)$, which will then be used as input to of the second layer to update $\mathbf{W}^2$. After all the layers are trained, we can finetune all the parameters of DBN with respect to a proxy for the DBN log- likelihood, or with respect to a supervised training criterion by adding a classifier such as softmax function on top of DBN.

A deep Boltzmann machine (DBM) [51] is another kind of deep generative model. Figure 11.1(d) gives an illustration of a 3 hidden layer DBM. Unlike DBN, it is an entirely undirected model. Unlike RBM, the DBM has several layers of latent variables (RBMs have just one). Within each layer, each of the variables are mutually independent, conditioned on the variables in the neighboring layers. In the case of a deep Boltzmann machine with one visible layer $\mathbf{v}$, and l hidden layers, $\mathbf{h}^1$, $\mathbf{h}^2$ and $\mathbf{h}^l$, the joint probability is given by:

$$P(\mathbf{v}, \mathbf{h}^1, \mathbf{h}^2, \ldots, \mathbf{h}^n) = \frac{1}{Z} \exp(-E(\mathbf{v}, \mathbf{h}^1, \mathbf{h}^2, \ldots, \mathbf{h}^n)) \tag{11.8}$$

where the DBM energy function is defined as:

$$E(\mathbf{v}, \mathbf{h}^1, \mathbf{h}^2, \ldots, \mathbf{h}^n) = -(\sum_{k=0}^{l-1} \mathbf{h}^k \mathbf{W}^k \mathbf{h}^{k+1}) - \sum_k \mathbf{b}^k \mathbf{h}^k \tag{11.9}$$

and $\mathbf{v} = \mathbf{h}^0$, $\mathbf{W}^k$ is weight matrix to capture the interaction between $\mathbf{h}^k$ and $\mathbf{h}^{k+1}$ and $\mathbf{b}^k$ is the bias.

The conditional distribution over one DBM layer given the neighboring layers is factorial. In the example of the DBM with two hidden layers, these distributions are $P(\mathbf{v}|\mathbf{h}^1)$, $P(\mathbf{h}^1|\mathbf{v}, \mathbf{h}^2)$ and $P(\mathbf{h}^2|\mathbf{h}^1)$. The distribution over all hidden layers generally does not factorize because of interactions between layers. In the example with two hidden layers, $P(\mathbf{h}^1, \mathbf{h}^2|\mathbf{v})$ does not factorize due to the interaction weights $\mathbf{W}^1$ between $\mathbf{h}^1$ and $\mathbf{h}^2$ which render these variables mutually dependent. Therefore, sampling from $P(\mathbf{h}^1, \mathbf{h}^2|\mathbf{v})$ is difficult while training of DBM using gradient ascent methods require to sample from $P(\mathbf{h}^1, \mathbf{h}^2|\mathbf{v})$. To solve this problem, we use mean-filed approximation to approximate $P(\mathbf{h}^1, \mathbf{h}^2|\mathbf{v})$. Specifically, we define

$$Q(\mathbf{h}^1, \mathbf{h}^2) = \prod_j Q(h_j^1|\mathbf{v}) \prod_k Q(h_k^2|\mathbf{v}) \tag{11.10}$$

The mean field approximation attempts to find a member of this family of distributions that best fits the true posterior $P(\mathbf{h}^1, \mathbf{h}^2|\mathbf{v})$ by minimizing KL-divergence between $Q(\mathbf{h}^1, \mathbf{h}^2)$ and $P(\mathbf{h}^1, \mathbf{h}^2|\mathbf{v})$. With the approximation, we can easily sample $\mathbf{h}^1$ and $\mathbf{h}^2$ from $Q(\mathbf{h}^1, \mathbf{h}^2)$ and then update the parameters using gradient ascents with these samples [51].

11.2.2 RBM for Real-Valued Data

In many real world applications such as image and audio modeling, the input features $\mathbf{v}$ are often real-valued data. Thus, it is important to extend RBM for modeling real-valued inputs. There are many variants of RBM which defines the probability over real-valued data such as Gaussian-Bernoulli RBMs [69], Mean and variance RBM [22] and Spike and Slab RBMs [8].

Gaussian-Bernoulli RBM (GBM) is the most common way to handle real-valued data, which has binary hidden units and real-valued visible units. It assumes the conditional distribution over the visible units being a Gaussian distribution whose mean is a function of the hidden units. Under this assumption, GRBM defines a joint probability over $\mathbf{v}$ and $\mathbf{h}$ as in Eq.(11.1) with the energy function given as

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{h}^T \mathbf{W}(\mathbf{v} \odot \boldsymbol{\beta}) - \mathbf{b}^T \mathbf{h} - \frac{1}{2}\mathbf{v} - \mathbf{c}^T(\boldsymbol{\beta} \odot (\mathbf{v} - \mathbf{c})) \tag{11.11}$$

where $\boldsymbol{\beta} \in \mathbb{R}^{m \times 1}$ is the precision vector with the i-th element β_i being the precision of v_i. $\odot$ is the Hadamard operation. Then the conditional probability of $P(\mathbf{v}|\mathbf{h})$ and $P(\mathbf{h}|\mathbf{v})$ are

$$P(\mathbf{h}|\mathbf{v}) = \prod_{i=1}^{d} P(h_i|\mathbf{v}) = \prod_{i=1}^{d} \sigma(b_i + \sum_{j=1}^{m} W_{ij} v_i \beta_i) \tag{11.12}$$

$$P(\mathbf{v}|\mathbf{h}) = \prod_{j=1}^{m} P(v_j|\mathbf{h}) = \prod_{j=1}^{m} \mathcal{N}(v_j|b_j + \sum_{i=1}^{d} W_{ij}h_i, \beta_i^{-1}) \tag{11.13}$$

where $\mathcal{N}(v_j|b_j + \sum_{i=1}^{d} W_{ij}h_i, \beta_i^{-1})$ is the Gaussian distribution with mean $b_j + \sum_{i=1}^{d} W_{ij}h_i$ and variance β_i^{-1}.

While the GRBM has been the canonical energy model for real-valued data, it is not well suited to the statistical variations present in some types of real-valued data, especially natural images [31]. The problem is that much of the information content present in natural images is embedded in the covariance between pixels rather than in the raw pixel values. To solve these problems, alternative models have been proposed that attempt to better account for the covariance of real-valued data. Mean and Covariance RBM (mcRBM) is one of the alternatives. The mcRBM uses its hidden units to independently encode the conditional mean and covariance of all observed units. Specifically, the hidden layer of mcRBM is divided into two groups of units: binary mean units $\mathbf{h}^{(m)}$ and binary covariance units $\mathbf{h}^{(c)}$. The energy function of mcRBM is defined as the combination of two energy functions:

$$E_{mc}(\mathbf{v}, \mathbf{h}^{(m)}, \mathbf{h}^{(c)}) = E_m(\mathbf{v}, \mathbf{h}^{(m)}) + E_c(\mathbf{v}, \mathbf{h}^{(c)}) \tag{11.14}$$

where $E_m(\mathbf{v}, \mathbf{h}^{(m)})$ is the standard Gaussian-Bernoulli energy function defined in Eq.(11.11), which models the interaction between real-valued $\mathbf{v}$ input and hidden units $\mathbf{h}^{(m)}$; and $E_c(\mathbf{v}, \mathbf{h}^{(c)})$ models the conditional covariance information, which is given as

$$E_c(\mathbf{v}, \mathbf{h}^{(c)}) = \frac{1}{2} \sum_j h_j^{(c)} (\mathbf{v}^T \mathbf{r}^{(j)})^2 - \sum_j b_j^{(c)} h_j^{(c)} \tag{11.15}$$

The parameter $\mathbf{r}^{(j)}$ corresponds to the covariance weight vector associated with $h_j^{(c)}$ and $\mathbf{b}^{(c)}$ is a vector of covariance offsets.

11.3 AutoEncoder

An autoencoder (AE) is a neural network trained to learn latent representation that is good at reconstructing it's input [4]. Generally, an autoencoder is composed of two parts, i.e., an encoder $f(\cdot)$ and a decoder $g(\cdot)$. An illustration of autoencoder is shown in Figure 11.2(a). The encoder maps the input $\mathbf{x} \in \mathbb{R}^m$ to latent representation $\mathbf{h} \in \mathbb{R}^d$ as $\mathbf{h} = f(\mathbf{x})$ and $f(\cdot)$ is usually a one layer neural network, i.e., $f(\mathbf{x}) = s(\mathbf{W}\mathbf{x} + \mathbf{b})$, where $\mathbf{W} \in \mathbb{R}^{d \times m}$ and $\mathbf{b} \in \mathbb{R}^d$ are the weights and bias of the encoder. $s(\cdot)$ is a non-linear function such as sigmoid and tanh. A decoder maps back the latent representation $\mathbf{h}$ into a reconstruction $\tilde{\mathbf{x}} \in \mathbb{R}^m$ as $\tilde{\mathbf{x}} = g(\mathbf{h})$ and $g(\cdot)$ is given as $g(\mathbf{h}) = s(\mathbf{W}'\mathbf{h} + \mathbf{b}')$,

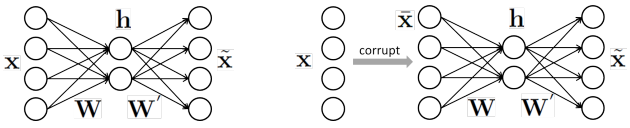
where $\mathbf{W}^{'} \in \mathbb{R}^{m \times d}$ and $\mathbf{b} \in \mathbb{R}^m$ are the weights and bias of the decoder. Note that the prime symbol does not indicate matrix transposition. The parameters of autoencoder, i.e., $\theta = \{\mathbf{W}, \mathbf{b}, \mathbf{W}', \mathbf{b}'\}$ are optimized to minimize the reconstruction error. Depending on the appropriate distribution assumptions of the input, the reconstruction error can be measured in many ways. The most widely used reconstruction error is the squared error $\mathcal{L}(\mathbf{x}, \tilde{\mathbf{x}}) = \|\mathbf{x} - \tilde{\mathbf{x}}\|_2^2$. Alternatively, if the input is interpreted as either bit vectors or vectors of bit probabilities, cross-entropy of the reconstruction can be used

$$\mathcal{L}_H(\mathbf{x}, \tilde{\mathbf{x}}) = -\sum_{k=1}^{d}[\mathbf{x}_k \log \tilde{\mathbf{x}}_k + (1 - \mathbf{x}_k \log(1 - \tilde{\mathbf{x}}_k))] \tag{11.16}$$

By training an autoencoder that is good at reconstructing input data, we hope that the latent representation $\mathbf{h}$ can capture some useful features. The identity function seems a particularly trivial function to be trying to learn, which doesn't result in useful features. Therefore, we need to add constraint to autoencoer to avoid trivial solution and learn useful features.

The autoencoder can be used to extract useful features by enforcing $\mathbf{h}$ to have smaller dimension than $\mathbf{x}$, i.e., $d < m$. An autoencoder whose latent dimension is less than the input dimension is called undercomplete autoencoder. Learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data [15]. In other words, the latent representation $\mathbf{h}$ is a distributed representation which captures the coordinates along the main factors of variation in the data [15]. This is similar to the way that the projection on principal components would capture the main factors of variation in the data. Indeed, if there is one linear hidden layer, i.e., no activation function applied, and the mean squared error criterion is used to train the network, then the d hidden units learn to project the input in the span of the first d principal components of the data. If the hidden layer is non-linear, the auto-encoder behaves differently from PCA, with the ability to capture multi-modal aspects of the input distribution.

Another choice is to constraint $\mathbf{h}$ to have larger dimension than $\mathbf{x}$, i.e., $d > m$. An autoencoder whose latent dimension is larger than the input dimension is called overcomplete autoencoder. However, due to the large dimension, the encorder and decoder are given too much capacity. In such cases, even a linear encoder and linear decoder can learn to copy the input to the output without learning anything useful about the data distribution. Fortunately, we can still discover interesting structure, by imposing other constraints on the network. One most widely used constraint is the sparsity constraint on $\mathbf{h}$. An overcomplete autoencoder with sparsity constraint is called sparse autoencoder, which will be discussed next.



(a) Autoencoder (b) Denoising AE

FIGURE 11.2: An illustration of autoencoder and denoising autoencoder

11.3.1 Sparse Autoencoder

A sparse autoencoder is an overcomplete authoencorder which tries to learn sparse overcomplete codes that are good at reconstruction [43]. A sparse over-complete representation can be viewed as an alternative "compressed" representation: it has implicit straightforward compressibility due to the large number of zeros rather than an explicit lower dimensionality. Given the training data $\mathbf{X} \in \mathbb{R}^{m \times N}$, the objective function is given as

$$\min_{\mathbf{W},\mathbf{b},\mathbf{W}',\mathbf{b}'} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\mathbf{x}_i, \tilde{\mathbf{x}}_i) + \alpha \Omega(\mathbf{h}_i) \tag{11.17}$$

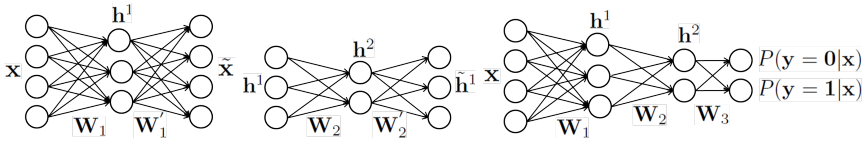
where N is number of training instances. $\mathbf{x}_i$ is the i-th training instance, $\mathbf{h}_i$ and $\tilde{\mathbf{x}}_i$ are the corresponding latent representation and reconstructed features. $\Omega(\mathbf{h}_i)$ is the sparsity regularizer to make $\mathbf{h}_i$ sparse and α is a scalar to control the sparsity. Many sparsity regularizer can be adopted. One popularly used is the ℓ_1-norm, i.e., $\Omega(\mathbf{h}_i) = \|\mathbf{h}_i\|_1 = \sum_{j=1}^{d} |h_i(j)|$. However, the ℓ_1-norm is non-smooth and not appropriate for gradient descent. An alternative is to use the smooth sparse constraint based on KL-divergence. Let ρ_j, $j = 1, \ldots, d$ be the average activation of hidden unit j (averaged over the training set) as

$$\rho_j = \frac{1}{N} \sum_{i=1}^{N} \mathbf{h}_i(j) \tag{11.18}$$

The essential idea is to enforce ρ_j to be close to ρ, where ρ is a small value close to zero (say $\rho = 0.05$). By enforcing ρ_j be close to ρ, we would like the average activation of each hidden neuron j to be close to 0.05 (say). This constraint is satisfied when the hidden units activations are mostly near 0. To achieve that ρ_j is close to ρ, we can use the KL-divergence as

$$\sum_{j=1}^{d} KL(\rho||\rho_j) = \sum_{j=1}^{d} \rho \log \frac{\rho}{\rho_j} + (1-\rho) \log \frac{1-\rho}{1-\rho_j} \tag{11.19}$$

$KL(\rho||\rho_j)$ is a convex function with it's minimum of when $\rho_j = \rho$. Thus, minimizing this penalty term has the effect of causing ρ_j to be close to ρ, which achieves the sparse effect.



(a) 1st Autoencoder (b) 2nd Autoencoder (c) Finetuning

FIGURE 11.3: An illustration of 2-layer stacked autoencoder

11.3.2 Denoising Autoencoder

The aforementioned autoencoders add constraints on latent representations to learn useful features. Alternatively, denoising autoencoder uses the denoising criteria to learn useful features. In order to force the hidden layer to discover more robust features and prevent it from simply learning the identity, denoising autoencoder train the autoencoder to reconstruct the input from a corrupted version of it [63]. An illustration of denoising autoencoder is shown in Figure 11.2(b). In the figure, the clean data $\mathbf{x}$ is first corrupted as a noisy data $\bar{\mathbf{x}}$ by means of a stochastic mapping $q_D(\bar{\mathbf{x}}|\mathbf{x})$. The corrupted data $\bar{\mathbf{x}}$ is then used as input to an autoencoder, which outputs the reconstructed data $\tilde{\mathbf{x}}$. The training objective of a denoising autoencoder is then to make reconstructed data $\tilde{\mathbf{x}}$ close to the clean data $\mathbf{x}$ as $\mathcal{L}(\mathbf{x}, \tilde{\mathbf{x}})$.

There are many choices of the stochastic mapping such as (1) Additive isotropic Gaussian noise (GS): $\bar{\mathbf{x}}|\mathbf{x} \sim N(\mathbf{x}, \sigma\mathbf{I})$; It is a very common noise model suitbale for real valued inputs(2) Masking noise (MN): a fraction ν of the elements of x (chosen at random for each example) is forced to 0; and (3) Salt-and-pepper noise (SP): a fraction ν of the elements of $\mathbf{x}$ (chosen at random for each example) is set to their minimum or maximum possible value (typically 0 or 1) according to a fair coin flip. The masking noise and salt-and-pepper noise are natural choices for input domains which are interpretable as binary or near binary such as black and white images or the representations produced at the hidden layer after a sigmoid squashing function [63].

11.3.3 Stacked Autoencoder

Denoising autoencoders can be stacked to form a deep network by feeding the latent representation of the DAE found on the layer below as input to the current layer as shown in Figure 11.3, which are generally called stacked denosing autoencoder (SDAE). The unsupervised pre-training of such an architecture is done one layer at a time. Each layer is trained as a DAE by minimizing the error in reconstructing its input. For example, in Figure 11.3(a), we train the first layer autoencoder. Once the first layer is trained, we can train the 2nd layer with the latent representation of the fist autoencoder, i.e., $\mathbf{h}^1$, as input. This is shown in Figure 11.3(b). Once all layers are pre-trained, the network goes through a second stage of training called fine-tuning, which are typically to minimize prediction error on a supervised task. For fine-tuning,

we first add a logistic regression layer on top of the network as shown in Figure 11.3(c) (more precisely on the output code of the output layer). We then train the entire network as we would train a multilayer perceptron. At this point, we only consider the encoding parts of each auto-encoder. This stage is supervised, since now we use the target class during training.

11.4 Convolutional Neural Networks

The Convolutional Neural Network (CNN or ConvNet) has achieved great success in many computer vision task such as image classification [32], segmentation [36] and video action recognition [55]. The specially designed architecture of CNN are very powerful in extracting visual features from images, which can be used for various tasks. An example of a simplified CNN is shown in Figure 11.4. It is comprised of three basic types of layers, which are convolutional layers for extracting translation-invariant features from images, pooling layers for reducing the parameters and fully-connected layers for classification tasks. CNNs are mainly formed by stacking these layers together. Recently, dropout layers [56] and residual layers [19] are also introduced to prevent CNN from overfitting and to ease the training of deep CNNs, respectively. Next, we will introduce the basic buliding blocks of CNNs and how CNN can be used for feature learning.

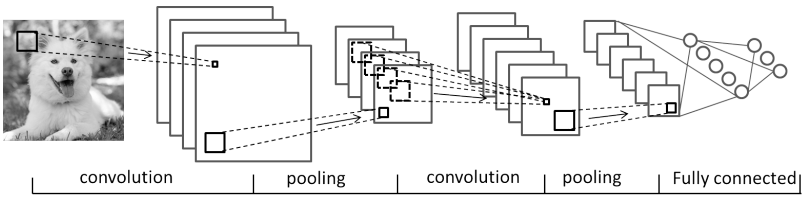


FIGURE 11.4: An illustration of CNN

The Convolutional Layer : As the name implies, the Conv layer is the core building block of a CNN. The essential idea of a Conv layer comes to the observation that natural images have the property of being "stationary", which means that the statistics of one part of the image are the same as any other part. For example, a dog can appear in any locations of an image. This suggests that the dog feature detector that we learn at one part of the image can also be applied to other parts of the image to detect dogs, and we can use the same features at all locations. More precisely, having learned features over small (say 3x3) patches sampled randomly from the larger image, we can then apply this learned 3x3 feature detector anywhere in the image. Specifically, we can take the learned 3x3 features and "convolve" them with the larger image, thus obtaining a different feature activation value at each location in

the image. The feature detector is called a filter or kernel in ConvNet and the features obtained is called a feature map. Figure 11.5 gives an example of convolution operation with the input as the 5x5 matrix and the kernel as the 3x3 matrix. The 3x3 kernel slides over the 5x5 matrix from left to right and top to down, which generates the feature map shown on the right. The convolution is done by multiplying the kernel with the sub-patch of the input feature map and then sum together. For example, the calculation of the gray sub-patch in 5x5 matrix with the kernel is given in the figure. There

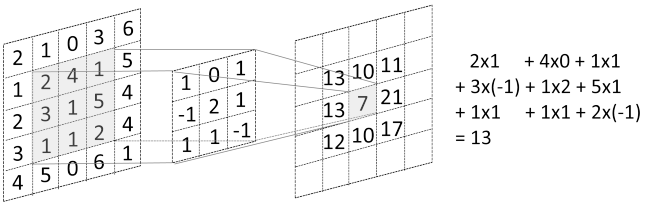


FIGURE 11.5: An illustration of convolution operation

are three parameters in a Conv layer, i.e., the *depth*, *stride* and *zero-padding*. *Depth* corresponds to the number of filters we would like to use. A Conv layer can have many filters each learning to look for something different in the input. For example, if the first Conv layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of color. In the simple ConvNet shown in Figure 11.4, the depth of the first convolution and second convolution layers are 4 and 6, respectively. *Stride* specifies how many pixels we skip when we slide the filter over the input feature map. When the stride is 1 then we move the filters one pixel at a time as shown in Figure 11.5. When the stride is 2 then the filters jump 2 pixels at a time as we slide them around. This will produce smaller output volumes spatially. It will be convenient to pad the input volume with zeros around the border, which is called *zero-padding*. The size of this zero-padding is a hyperparameter. The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes. Let the input volume be $W \times H \times K$, where W and H are width and height of the feature map and K is the number of feature maps. For example, for a color image with RGB channels, we have $K = 3$. Let the the receptive field size (filter size) of the Conv Layer be F, number of filters be $\tilde{K}$, the stride with which they are applied be S, and the amount of zero padding used on the border be P, then the output volume after convolution is $\tilde{W} \times \tilde{H} \times \tilde{K}$, where $\tilde{W} = (W - F + 2P)/S + 1$ and $\tilde{H} = (H - F + 2P)/S + 1$. For example, for a $7 \times 7 \times 3$ input and a $4 \times 3 \times 3$ filter with stride 1 and pad 0 we would get a $5 \times 5 \times 4$ output.

The convolution using filters is linear operation. After the feature maps are obtained in a Conv layer, a nonlinear activation function will be applied on these feature maps to learn non-linear features. Rectified linear unit (ReLU) is the most widely used activation function for ConvNet, which is demonstrated

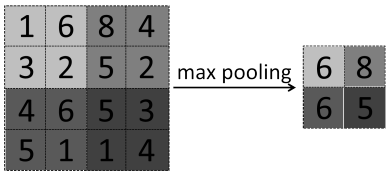


FIGURE 11.6: An illustration of max pooling operation

to be effective in alleviating the gradient vanishing problem. A rectifier is defined as $f(x) = \max(0, x)$.

The Pooling Layer: Pooling layers are usually periodically inserted in-between successive Conv layers in a CNN. They aim to progressively reduce the spatial size of the representation, which can help reduce the amount of parameters and computation in the network, and hence to also control over-fitting. The pooling layer operates independently over each activation map in the input, and scales its dimensionality using the *max* function. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2, which downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every max operation would in this case be taking a max over 4 numbers and the maximum value of the 4 numbers will go to next layer. An example of max pooling operation is shown given in Figure 11.6. Hence, during the forward pass of a pooling layer it is common to keep track of the index of the max activation (sometimes also called the switches) so that gradient routing is efficient during backpropagation.

Though max pooling is the most popular pooling layer, a CNN can also contain general-pooling. General pooling layers are comprised of pooling neurons that are able to perform a multitude of common operations including L1/L2-normalization, and average pooling. An example of max pooling

The Fully Connected Layer : Neurons in a fully connected layer have full connections to all activations in the previous layer, as shown in Figure 11.4. The fully connected layers are put at the end of a CNN architecture, i.e., after several layers of Conv layer and max pooling layers. With the high level features extracted by the previous layers, fully connected layers will then attempt to produce class scores from the activations, to be used for classification. The output of the fully connected layer will then be put in a softmax for classification. It is also suggested that ReLu may be used as the activation function in fully connected layer as to improve performance.

11.4.1 Transfer Feature Learning of CNN

In practice, training an entire Convolutional Network from scratch (with random initialization) is rare as (1) it is very time consuming, requires many computation resources and (2) it is relatively rare to have a dataset of sufficient size to train a ConvNet. Therefore, instead, it is common to pre-train a

ConvNet on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an initialization or a fixed feature extractor for the task of interest [53]. There are mainly two major Transfer Learning scenarios, which are listed as follows:

- ConvNet as a fixed feature extractor: In this scenario, we take a ConvNet pretrained on ImageNet, remove the last fully-connected layer, then treat the rest of the ConvNet as a fixed feature extractor for the new dataset. With the extracted features, we can train a linear classifier such as Linear SVM or logistic regression for the new dataset. This is usually used when the new dataset is small and similar to original dataset. For such dataset, training or finetuning a ConvNet is not practical as ConvNet are prone to overfitting to small dataset. Since the new dataset is similar to original dataset, we can expect higher-level features in the ConvNet to be relevant to this dataset as well.
- Fine-tuning the ConvNet: The second way is to not only replace and retrain the classifier on top of the ConvNet on the new dataset, but to also fine-tune the weights of the pretrained network using backpropagation. The essential idea of fine-tuning is that the earlier features of a ConvNet contain more generic features (e.g. edge detectors or color blob detectors) that should be useful to many tasks, but later layers of the ConvNet becomes progressively more specific to the details of the classes contained in the original dataset. If the new dataset is large enough, we can fine-tune all the layers of the ConvNet. If the new dataset is small but different from original dataset, then we can keep some of the earlier layers fixed (due to overfitting concerns) and only fine-tune some higher-level portion of the network.

11.5 Word Embedding and Recurrent Neural Networks

Word embedding and recurrent neural networks are the state-of-the-art deep learning models for natural language processing tasks. Word embedding learns word representation and recurrent neural network utilizes word embedding for sentence or document feature learning. Next, we introduce the details of word embedding and recurrent neural networks.

11.5.1 Word Embedding

Word embedding, or distributed representation of words, is to represent each word as a low dimensional dense vector such that the vector representation of words can capture synthetic and semantic meanings of words. The

low dimensional representation also can alleviate the curse of dimensionality and data sparsity problems suffered by traditional representations such as bag-of-words and N-gram [66]. The essential idea of word embedding is the distributional hypothesis that "you shall know a word by the company it keeps" [13]. This suggests that a word has close relationships with its neighboring words. For example, the phrases *win the game* and *win the lottery* appear very frequently; thus the pair of words *win* and *game* and the pair of words *win* and *lottery* could have very close relationship. When we are only given the word *win*, we would highly expect the neighboring words to be words like *game* or *lottery* instead of words as *light* or *air*. This suggests that a good word representation should be useful for predicting its neighboring words, which is the essential idea of Skip-gram [41]. In other words, the training objective of the Skip-gram model is to find word representations that are useful for predicting the surrounding words in a sentence or a document. More formally, given a sequence of training words $w_1, w_2, \ldots, w_T$, the objective of the Skip-gram model is to maximize the average log probability

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log P(w_{t+j}|w_t) \tag{11.20}$$

where c is the size of the training context (which can be a function of the center word w_t). Larger c results in more training examples and thus can lead to a higher accuracy, at the expense of the training time. The basic Skip-gram formulation defines $P(w_{t+j}|w_t)$ using the softmax function:

$$P(w_O|w_I) = \frac{\exp(\mathbf{u}_{w_O}^T \mathbf{v}_{w_I})}{\sum_{w=1}^{W}\exp({\mathbf{u}_w}^T \mathbf{v}_{w_I})} \tag{11.21}$$

where $\mathbf{v}_w$ and $\mathbf{u}_w$ are the "input" and "output" representations of w, and W is the number of words in the vocabulary. Learning the representation is usually done by gradient descent. However, Eq.(11.21) is impractical because the cost of computing $\nabla \log P(w_O|w_I)$ is proportional to W, which is often large. One way of making the computation more tractable is to replace the softmax in Eq.(11.21) with a hierarchical softmax. In hierarchial softmax, the vocabulary is represented as a Huffman binary tree with words as leaves. With Huffman tree, the probability of $P(w_O|w_I)$ is the probability of walking the path from root node to leaf node w_O given the word w_I, which is calculated as decision making in each node along the path with simple function. The Huffman trees assign short binary codes to frequent words, and this further reduces the number of output units that need to be evaluated. Another alternative to make the computation tractable is negative sampling [41]. The essential idea of negative sampling is that w_t should be more similar with its neighboring words say w_{t+j} than randomly sampled words. Thus, the objective function of negative sampling is to maximize the similarity between w_t and w_{t+j} while minimize the similarity between w_t and randomly sampled

words. With negative sampling, Eq.(11.21) is approximated as

$$\log \sigma(\mathbf{u}_{W_O}^T \mathbf{v}_{w_I}) + \frac{1}{K} \sum_{i=1}^{K} \log \sigma(-\mathbf{u}_{W_i}^T \mathbf{v}_{w_I}) \tag{11.22}$$

where K is the number of negative words sampled for each input word w_I. It is found that skip-gram with negative sampling is equivalent to implicitly factorizing a word-context matrix, whose cells are the pointwise mutual information (PMI) of the respective word and context pairs, shifted by a global constant [34].

Instead of using the center words to predict the context (surrounding words in a sentence), Continuous Bag-of-Words Model (CBOW) predicts the current word based on the context. More precisely, CBOW uses each current word as an input to a log-linear classifier with continuous projection layer, and predict words within a certain range before and after the current word [39]. The objective function of CBOW is to maximize the following log-likelihood function

$$\frac{1}{T} \sum_{t=1}^{T} \log P(w_t | w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c}) \tag{11.23}$$

and $P(w_t | w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c})$ is defined as

$$P(w_t | w_{t-c}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+c}) = \frac{\exp({\mathbf{u}_{w_t}}^T \tilde{\mathbf{v}}_t)}{\sum_{w=1}^{W} \exp({\mathbf{u}_w}^T \tilde{\mathbf{v}}_t)} \tag{11.24}$$

with $\tilde{\mathbf{v}}_t$ is the average representation of the contexts of w_t, i.e., $\tilde{\mathbf{v}}_t = \frac{1}{2c} \sum_{-c \leq j \leq c, j \neq 0} \mathbf{v}_{t+j}$.

Methods like skip-gram may do better on the analogy task, but they poorly utilize the statistics of the corpus since they train on separate local context windows instead of on global co-occurrence counts. Based on this observation, GloVe proposed in [46] uses a specific weighted least squares model that trains on global word-word co-occurrence counts and thus makes efficient use of statistics. The objective function of GloVe is given as

$$\min \sum_{i,j} f(X_{ij}) (\mathbf{w}_i^T \tilde{\mathbf{w}}_j - \log X_{ij})^2 \tag{11.25}$$

where X_{ij} tabulate the number of times word j occurs in the context of word i. $\mathbf{w}_i \in \mathbb{R}^d$ is word representation of w_i and $\tilde{\mathbf{w}}_j$ is separate context word vector. $f()$ is the weighting function.

Word embedding can capture syntactic and semantic meanings of words. For example, it is found that vec(queen) is the closest vector representation to vec(king) - vec(man) + vec(woman), which implies that word representation learned by Skip-gram encodes semantic meanings of words. Word embedding can also be used for document representation by averaging the word vectors of words appearing in a document as the vector representation of the documents.

Following the distributional representation idea of word embedding, many network embedding algorithms are proposed. The essential idea of network embedding is to learn vector representations of network nodes that are good at predicting the neighboring nodes.

Since word representation learned by word embedding algorithms are low-dimensional dense vectors that capture semantic meanings, they are widely used as preprocessing step in deep learning methods such as recurrent neural network and recursive neural networks. Each word will be mapped to vector representation before it is used as input to deep learning models.

11.5.2 Recurrent Neural Networks

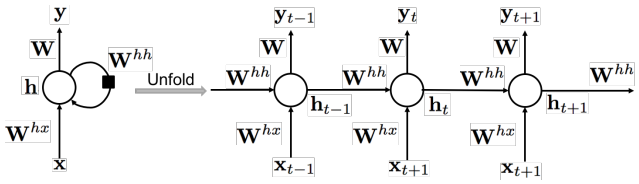


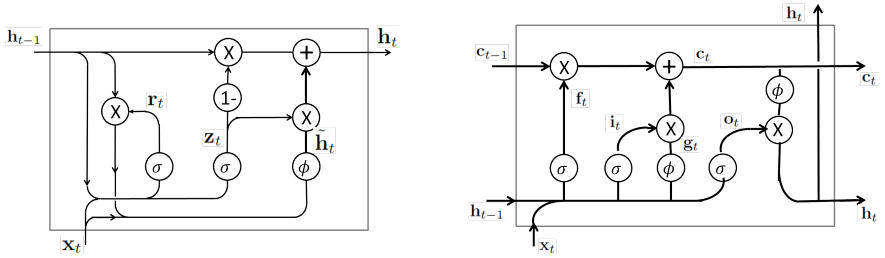
FIGURE 11.7: An illustration of RNN

Recurrent neural networks (RNN) are powerful concepts that allow the use of loops within the neural network architecture to model sequential data such as sentences and videos. Recurrent networks take as input a sequence of inputs, and produce a sequence of outputs. Thus, such models are particularly useful for sequence-to-sequence learning.

Figure 11.7 gives an illustration of the RNN architecture. The left part of the figure shows a folded RNN, which has a self-loop, i.e., the hidden state $\mathbf{h}$ is used to update itself given an input $\mathbf{x}$. To better show how RNN works, we unfold the RNN as a sequential structure, which is given in the right part of Figure 11.7. The RNN takes a sequence, $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t, \ldots, \mathbf{x}_T$ as input, where at each step t, $\mathbf{x}_t$ is a d-dimensional feature vector. For example, if the input is a sentence, then each word w_i of the sentence is represented as a vector $\mathbf{x}_i$ using word embedding methods such as Skip-gram. At each time-step t, the output of the previous step, $\mathbf{h}_{t-1}$, along with the next word vector in the document, $\mathbf{x}_t$, are used to update the hidden state $\mathbf{h}_t$ as

$$\mathbf{h}_t = \sigma(\mathbf{W}^{hh}\mathbf{h}_{t-1} + \mathbf{W}^{hx}\mathbf{x}_t) \tag{11.26}$$

where $\mathbf{W}^{hh} \in \mathbb{R}^{d\times d}$ and $\mathbf{W}^{hx} \in \mathbb{R}^{d\times d}$ are the weights for inputs $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$, respectively. *The hidden states $\mathbf{h}_t$ is the feature representation of the sequence up to time t for the input sequence.* The initial state $\mathbf{h}_0$ are usually initialized as all 0. Thus, we can utilize $\mathbf{h}_t$ to perform various tasks such as sentence completion and document classification. For example, for sentence completion task, we are given a partial sentence as "The weather is " and we want to



(a) One GRU Cell (b) One LSTM Cell

FIGURE 11.8: An illustration of GRU and LSTM Cells

predict the next word. We can predict the next word as

$$\mathbf{y}_t = softmax(\mathbf{W}\mathbf{h}_t + \mathbf{b}), \quad y_t = \arg\max \mathbf{y}_t \tag{11.27}$$

where $\mathbf{W} \in \mathbb{R}^{V \times d}$ is the weights of the softmax function with V being the size of the vocabulary and b is the bias term. $\mathbf{y}_t$ is the predicted probability vector and y_t is the predicted label. We can think of RNN models the likelihood probability as $P(\mathbf{y}_t|\mathbf{x}_1, \ldots, \mathbf{x}_t)$.

Training of RNN is usually done using Backpropagation Through Time (BPTT), which back-propagates the error from time t to time 1 [70].

11.5.3 Gated Recurrent Unit

Though in theory, RNN is able to capture long-term dependency, in practice, the old memory will fade away as the sequence becomes longer. To making it easier for RNNs to capture long-term dependencies, Gated recurrent units (GRU) [7] are designed in a manner to have more persistent memory. Unlike RNN, which uses simple affine transformation of $\mathbf{h}_{t-1}$ and $\mathbf{h}_t$ followed by *tanh* to update $\mathbf{h}_t$, GRU introduces Reset Gate to determine if it want to forget past memory and Update Gate to control if new inputs are introduced to $\mathbf{h}_t$. The mathematical equations of how this is achieved are given as follows and an illustration of a GRU cell is shown in Figure 11.8(a):

$$\begin{aligned}
\mathbf{z}_t &= \sigma(\mathbf{W}_{zx}\mathbf{x}_t + \mathbf{W}_{zh}\mathbf{h}_{t-1} + \mathbf{b}_z) && \text{(Update gate)} \\
\mathbf{r}_t &= \sigma(\mathbf{W}_{rx}\mathbf{x}_t + \mathbf{W}_{rh}\mathbf{h}_{t-1} + \mathbf{b}_r) && \text{(Reset gate)} \\
\tilde{\mathbf{h}}_t &= \tanh(\mathbf{r}_t \odot \mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t) && \text{(New memory)} \\
\mathbf{h}_t &= (1 - \mathbf{z}_t) \odot \tilde{\mathbf{h}}_t + \mathbf{z}_t \odot \mathbf{h}_{t-1} && \text{(Hidden state)}
\end{aligned} \tag{11.28}$$

From the above equation and Figure 11.8(a), we can treat GRU as four fundamental operational stages, i.e., new memory, update gate, reset gate and hidden state. A new memory $\tilde{h}_t$ is the consolidation of a new input word $\mathbf{x}_t$ with the past hidden state $\mathbf{h}_{t-1}$, which summarize this new word in light of the contextual past. The reset signal $\mathbf{r}_t$ is used to determining how important

$\mathbf{h}_{t-1}$ is to the summarization $\tilde{\mathbf{h}}_t$. The reset gate has the ability to completely diminish past hidden state if it finds that $\mathbf{h}_{t-1}$ is irrelevant to the computation of the new memory. The update signal $\mathbf{z}_t$ is responsible for determining how much of past state $\mathbf{h}_{t-1}$ should be carried forward to the next state. For instance, if $\mathbf{z}_t \approx 1$, then $\mathbf{h}_{t-1}$ is almost entirely copied out to $\mathbf{h}_t$. The hidden state $\mathbf{h}_t$ is finally generated using the past hidden input $\mathbf{h}_t$ and the new memory generated $\tilde{\mathbf{h}}_t$ with the control of update gate.

11.5.4 Long Short-Term Memory

Long-Short-Term-Memories, LSTMs [23], are another type variants of RNN, which can also capture long-term dependency. Similar to GRUs, a LSTM introduces more complex gates to control if it should accept new information or forget previous memory, i.e., input gate, forget gate, output gate and new memory cell. The update rules of LSTMs are given as follows:

$$\begin{aligned}
\mathbf{i}_t &= \sigma(\mathbf{W}_{ix}\mathbf{x}_t + \mathbf{W}_{ih}\mathbf{h}_{t-1} + \mathbf{b}_i) && \text{(Input gate)} \\
\mathbf{f}_t &= \sigma(\mathbf{W}_{fx}\mathbf{x}_t + \mathbf{W}_{fh}\mathbf{h}_{t-1} + \mathbf{b}_f) && \text{(Forget gate)} \\
\mathbf{o}_t &= \sigma(\mathbf{W}_{ox}\mathbf{x}_t + \mathbf{W}_{oh}\mathbf{h}_{t-1} + \mathbf{b}_o) && \text{(Output gate)} \\
\mathbf{g}_t &= \tanh(\mathbf{W}_{gx}\mathbf{x}_t + \mathbf{W}_{gh}\mathbf{h}_{t-1} + \mathbf{b}_g) && \text{(New memory cell)} \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t && \text{(Final memory cell)} \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t) && (11.29)
\end{aligned}$$

where $\mathbf{i}_t$ is the input gate, $\mathbf{f}_t$ is the forget gate, $\mathbf{o}_t$ is the forget fate, $\mathbf{c}_t$ is the memory cell state at t and $\mathbf{x}_t$ is the input features at t. $\sigma(\cdot)$ means the sigmoid function and $\odot$ denotes the Hadmard product. The main idea of the LSTM model is the memory cell $\mathbf{c}_t$, which records the history of the inputs observed up to t. $\mathbf{c}_t$ is a summation of – (1) the previous memory cell $\mathbf{c}_{t-1}$ modulated by a sigmoid gate $\mathbf{f}_t$, and (2) $\mathbf{g}_t$, a function of previous hidden states and the current input modulated by another sigmoid gate $\mathbf{i}_t$. The sigmoid gate $\mathbf{f}_t$ is to selectively forget its previous memory while $\mathbf{i}_t$ is to selectively accept the current input. $\mathbf{i}_t$ is the gate controlling the output. The illustration of a cell of LSTM at the time step t is shown in Figure 11.8(b).

11.6 Generative Adversarial Networks and Variational Autoencoder

In this section, we introduce two very popular deep generative models proposed recently, i.e., generative adversarial networks and variational autoencoder.

11.6.1 Generative Adversarial Networks

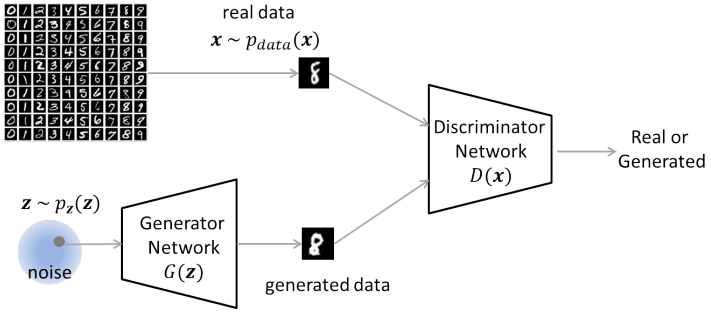


FIGURE 11.9: An illustration of the framework of GAN

Generative adversarial network (GAN) [16] is one of the most popular generative deep models. The core of a GAN is to play a min-max game between a discriminator D and a generator G, i.e., adversarial training. The discriminator D tries to differentiate if a sample is from real-world or generated by the generator while the generator G tries to generate samples that can fool the discriminator, i.e., make the discriminator believe that the generated samples are from real-world. Figure 11.9 gives an illustration of the framework of GAN. The generator takes a noise $\mathbf{z}$ sampled from a prior distribution $p_{\mathbf{z}}(\mathbf{z})$ as input, and maps the noise to the data space as $G(\mathbf{z};\theta_g)$. Typical choices of the prior $p(\mathbf{z})$ can be uniform distribution or Gaussian distribution. We also define a second multilayer perceptron $D(\mathbf{x};\theta_d)$ that outputs a single scalar. $D(\mathbf{x})$ represents the probability that $\mathbf{x}$ came from the real-world data rather than generated data. D is trained to maximize the probability of assigning the correct label to both training examples and samples from G. We simultaneously train G to minimize $\log(1 - D(G(\mathbf{z})))$. In other words, D and G play the following two-player minimax game with value function $V(D, G)$:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}(\mathbf{z})}}[\log(1 - D(G(\mathbf{z})))] \tag{11.30}$$

The training of GAN can be done using minibatch stochastic gradient descent training by updating the parameters of G and D alternatively. *After the model is trained unsupervisedly, we can treat the discriminator as a feature extractor* as: The first few layers of D is to extract features from $\mathbf{x}$ while the last few layers are to map the features to the probability that $\mathbf{x}$ is from real data. Thus, we can remove the last few layers, then the the output of D is the features extracted. In this sense, we treat GAN as unsupervised feature learning algorithms though the main purpose of GAN is to learn $p(\mathbf{x})$.

GAN is a general adversarial training framework, which can be used for various domains by designing different generator, discriminator and loss function [6, 65, 74]. For example, InfoGAN [6] learns disentangled representation by dividing the noise into two parts, i.e., disentangled codes $\mathbf{c}$ and incom-

pressible noise $\mathbf{z}$ so that the disentangled codes $\mathbf{c}$ can control the properties such as the identity and illumination of the images generated. SeqGAN [74] models the data generator as a stochastic policy in reinforcement learning and extends GAN for text generation.

11.6.2 Variational Autoencoder

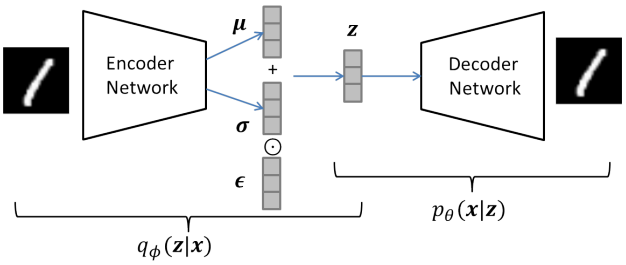


FIGURE 11.10: An illustration of the framework of VAE

Variational Autoencoder (VAE) [30] is a popular generative model for unsupervised representation learning. It can be trained purely with gradient-based methods. Typically, a VAE has a standard autoencoder component which encodes the input data into a latent code space by minimizing reconstruction error, and a Bayesian regularization over the latent space, which enforces the posterior of the hidden code vector matches a prior distribution. Figure 11.10 gives an illustration of an VAE. To generate a sample from the model, the VAE first draws a sample $\mathbf{z}$ from the prior distribution $p_\theta(\mathbf{z})$. The sample $\mathbf{z}$ is used as input to a differentiable generator network $g(\mathbf{z})$. Finally, $\mathbf{x}$ is sampled from a distribution $p_\theta(\mathbf{x}|g(\mathbf{z})) = p_\theta(\mathbf{x}|\mathbf{z})$. During the training, the approximate inference network, i.e., encoder network $q_\phi(\mathbf{z}|\mathbf{x})$ is then used to obtain $\mathbf{z}$ and $p_\theta(\mathbf{x}|\mathbf{z})$ is then viewed as a decoder network. The core idea of variational autoenoder is that they are trained by maximizing the variational lower bound $\mathcal{L}(\theta, \phi; \mathbf{x})$:

$$\mathcal{L}(\theta, \phi; \mathbf{x}) = -D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z})) + \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] \leq \log p_\theta(\mathbf{x}) \tag{11.31}$$

where $D_{KL}(q_\phi(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z}))$ is the KL divergence which measures the similarity of two distributions $q_\phi(\mathbf{z}|\mathbf{x})$ and $p_\theta(\mathbf{z})$. ϕ and θ are the variational parameters and generative parameters, respectively. We want to differentiate and optimize the lower bound w.r.t both the ϕ and θ. However, directly using gradient estimator for the above objective function will exhibits high variance. Therefore, VAE adopts the reparametric trick. That is, under certain mild conditions for a chosen approximate posterior $q_\phi(\mathbf{z}|\mathbf{x})$, the random variable $\tilde{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$ can be parameterized as

$$\tilde{z} = g_\phi(\epsilon, \mathbf{x}) \quad \text{with} \quad \epsilon \sim p(\epsilon) \tag{11.32}$$

where $g_\phi(\epsilon, \mathbf{x})$ is a differentiable transformation function of an noise variable ϵ. The nonparametric trick is also shown in the Figure 11.10. With this technique, the variational lower bound in Eq.(11.31) can be approximated as

$$\begin{aligned} L^A(\theta, \phi; \mathbf{x}) &= \frac{1}{L} \sum_{l=1}^{L} \log p_\theta(\mathbf{x}, \mathbf{z}^{(l)}) - \log q_\phi(\mathbf{z}^{(l)}|\mathbf{x}) \\ \tilde{z}^{(l)} &= g_\phi(\epsilon^{(l)}, \mathbf{x}) \quad \text{with} \quad \epsilon^{(l)} \sim p(\epsilon) \end{aligned} \tag{11.33}$$

Then the parameters can be learned via stochastic gradient descent efficiently. It is easy to see that the encoder is a feature extractor which learns latent representation for $\mathbf{x}$.

11.7 Discussion and Further Readings

We have introduced representative deep learning models for feature engineering. In this section, we'll discuss how they can be used for hierarchical representation learning and disentangled representation, and how they can be used for popular domains such as text, image and graph.

TABLE 11.1: Hierarchical and Disentangled Representation Learning

Method	Hierarchical Fea. Rep.	Disentangled Fea. Rep.
RBM/DBM	[59]	[48]
DBN	[33]	N/A
AE/DAE/SDAE	[37]	[28]
RNN/GRU/LSTM	[73, 68]	[54, 11]
CNN	[12, 14]	[49, 25]
GAN	[47]	[6, 38]
VAE	[75]	[72, 54]

Hierarchical Representation Generally, hierarchical representation is to learn features of hierarchy and further be able to combine top-down and bottom up processing of an image (or text). For instance, lower layers could support object detection by spotting low-level features indicative of object parts. Conversely, information about objects in the higher layers could resolve lower-level ambiguities in the image or infer the locations of hidden object parts. Features at different hierarchy may good for different tasks. The high-level features captures the main objects, resolve ambiguities and thus are good for classification while mid-level features kept many details and may be good for segmentation. Hierarchical feature representation is very common in deep learning models. We list some representative literature of how the introduced model can be used for hierarchical feature learning in Table 11.1

Disentangled Representation Disentangled representation is a popular way to learn explainable representation. The majority of existing representation learning framework learns representation $\mathbf{h} \in \mathbb{R}^{d\times 1}$ that is difficult to explain, i.e., the d-latent dimensions are entangled together and we don't know the semantic meaning of the d-dimensions. Instead, disentangled representation learning tries to disentangle the latent factors so we know the semantic meaning of the latent dimensions. For example, for handwritten digit such as MNIST dataset, we may want to disentangle the digit shape from writing style so that some part of $\mathbf{h}$ controls digit shapes while the other part represents writing style. The disentangled representation not only gives explanation for latent representation but also helps to generate controlled realistic images. For example, by changing the part of codes that controls digit shape, we can generate new images of target digit shape using generator with this new latent representation. Therefore, disentangled representation learning is attracting increasing attention. Table 11.1 also list some representative deep learning methods for disentangled representation learning. This is still a relatively new direction that needs further investigation.

TABLE 11.2: Deep Learning Methods for Different Domains

Method	Text	Image	Audio	Linked Data (Graph)
RBM/DBM	[58, 57]	[57]	[9]	[67]
DBN	[52]	[33]	[42]	N/A
AE/DAE/SDAE	[3]	[63]	[44]	[64]
CNN	[29]	[45, 19, 71]	[1]	[10]
Word2Vec	[41, 35]	N/A	N/A	[61, 66]
RNN/GRU/LSTM	[60, 40, 27]	[2]	[17, 50]	N/A
GAN	[74]	[6, 47]	[74]	N/A
VAE	[5, 26]	[30, 18]	[24]	N/A

Deep Feature Representation for Various Domains Many deep learning algorithms were initially developed for specific domains. For example, CNN is initially developed for image processing and Word2Vec is initially proposed for learning word representation. As the great success of these methods, they are further developed to be applicable to other domains. For example, in addition to image, CNN has also been successfully applied on texts, audio and linked data. Each domain has its own unique property. For example, text data are inherently sequential and graph data is non-i.i.d. Thus, directly applying CNN is impractical. New architecture are proposed to adapt CNN for these domains. The same holds for the other deep learning algorithms. Therefore, we summarize the application of the discussed deep learning models on four domains in Table 11.2. We encourage interested users to read these papers for further understanding.

Combining Deep Learning Models We have introduced various deep learning models, which can be applied for different domains. For example,

LSTM is mainly used for dealing sequential data such as texts while CNN is powerful for images. It is very common that we need to work on tasks which are related to different domains. In such cases, we can combine different deep learning models to propose a new framework that can be applifed for the task at the hand. For example, for information retrieve task that given a text query, we want to retrieve images marches the query. We will need to use LSTM or Word2Vec to learn representation that captures the semantic meanings of the query. At the same time, we need to use CNN to learn features that describe the image. We can then train LSTM and CNN in a way such that the similarity of the representations for the matched query-image pairs are maximized while the similarity of the representations for the non-marched query-image pairs are minimized. Another example is veido action recognition, where we want to classify the action of the video. Since the video is composed of frames and nearby frames have dependency. Thus, a vidao is inherently a sequential data and LSTM is a good fit for modeling such data. However, LSTM is not good at extracting images. Therefore, we will first need to use CNN to extract features from each frame of the video, which are then used as input to LSTM for learning representation of the video [68]. Similarity, for image captioning, we can use CNN to extract features and use LSTM to generate image caption based on the image features [71]. We just list a few examples and there are many other examples. In general, we can treat deep learning algorithms as feature extracting tools that can be used to extract features from certain domains. We can then design loss function on top of these deep learning algorithms for the problem we want to study. One thing to note is that, when we combine different models together, they are trained end-to-end. In other words, we don't train these models separately. In stead, we treat the new model as a unified model. This usually gives better performance than training each model separately and then combine them together.

Bibliography

[1] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on audio, speech, and language processing*, 22(10):1533–1545, 2014.

[2] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C. Lawrence Zitnick, and Devi Parikh. VQA: visual question answering. In *CVPR*, pages 2425–2433, 2015.

[3] Sarath Chandar AP, Stanislas Lauly, Hugo Larochelle, Mitesh Khapra, Balaraman Ravindran, Vikas C Raykar, and Amrita Saha. An autoencoder approach to learning bilingual word representations. In *NIPS*, pages 1853–1861, 2014.

[4] Yoshua Bengio et al. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[5] Samuel R Bowman, Luke Vilnis, Oriol Vinyals, Andrew M Dai, Rafal Jozefowicz, and Samy Bengio. Generating sentences from a continuous space. *CoNLL*, 2016.

[6] Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In *NIPS*, pages 2172–2180, 2016.

[7] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[8] Aaron Courville, James Bergstra, and Yoshua Bengio. A spike and slab restricted boltzmann machine. In *AISTAS*, pages 233–241, 2011.

[9] George Dahl, Abdel-rahman Mohamed, Geoffrey E Hinton, et al. Phone recognition with the mean-covariance restricted boltzmann machine. In *NIPS*, pages 469–477, 2010.

[10] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, pages 3844–3852, 2016.

[11] Emily Denton and Vighnesh Birodkar. Unsupervised learning of disentangled representations from video. 2017.

[12] Clement Farabet, Camille Couprie, Laurent Najman, and Yann LeCun. Learning hierarchical features for scene labeling. *IEEE TPAMI*, 35(8):1915–1929, 2013.

[13] John R Firth. A synopsis of linguistic theory, 1930-1955. 1957.

[14] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR*, pages 580–587, 2014.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[16] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *NIPS*, pages 2672–2680, 2014.

[17] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *ICASSP*, pages 6645–6649. IEEE, 2013.

[18] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*, 2015.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[20] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[21] Geoffrey E Hinton. Deep belief networks. *Scholarpedia*, 4(5):5947, 2009.

[22] Geoffrey E Hinton et al. Modeling pixel means and covariances using factorized third-order boltzmann machines. In *CVPR*, pages 2551–2558. IEEE, 2010.

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

[24] Wei-Ning Hsu, Yu Zhang, and James R. Glass. Learning latent representations for speech generation and transformation. volume abs/1704.04222, 2017.

[25] Wei-Ning Hsu, Yu Zhang, and James R. Glass. Unsupervised learning of disentangled and interpretable representations from sequential data. In *NIPS*, 2017.

[26] Zhiting Hu, Zichao Yang, Xiaodan Liang, Ruslan Salakhutdinov, and Eric P Xing. Toward controllable text generation. In *ICML*, 2017.

[27] Ozan Irsoy and Claire Cardie. Opinion mining with deep recurrent neural networks. In *EMNLP*, pages 720–728, 2014.

[28] Michael Janner, Jiajun Wu, Tejas Kulkarn, Ilker Yildirim, and Josh Tenenbaum. Learning to generalize intrinsic images with a structured disentangling autoencoder. In *NIPS*, 2017.

[29] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. In *ACL*, 2014.

[30] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[31] Alex Krizhevsky, Geoffrey E Hinton, et al. Factored 3-way restricted boltzmann machines for modeling natural images. In *AISTATS*, pages 621–628, 2010.

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1097–1105, 2012.

[33] Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Y Ng. Unsupervised learning of hierarchical representations with convolutional deep belief networks. *Communications of the ACM*, 54(10):95–103, 2011.

[34] Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. In *NIPS*, pages 2177–2185, 2014.

[35] Yang Li, Quan Pan, Tao Yang, Suhang Wang, Jiliang Tang, and Erik Cambria. Learning word representations for sentiment analysis. *Cognitive Computation*, 2017.

[36] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *CVPR*, pages 3431–3440, 2015.

[37] Jonathan Masci, Ueli Meier, Dan Cireşan, and Jürgen Schmidhuber. Stacked convolutional auto-encoders for hierarchical feature extraction. *Artificial Neural Networks and Machine Learning–ICANN 2011*, pages 52–59, 2011.

[38] Michaël Mathieu, Junbo Jake Zhao, Pablo Sprechmann, Aditya Ramesh, and Yann LeCun. Disentangling factors of variation in deep representation using adversarial training. In *NIPS*, pages 5041–5049, 2016.

[39] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[40] Tomas Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, pages 1045–1048, 2010.

[41] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119, 2013.

[42] Abdel-rahman Mohamed, George Dahl, and Geoffrey Hinton. Deep belief networks for phone recognition. In *Nips workshop on deep learning for speech recognition and related applications*, 2009.

[43] Andrew Ng. Sparse autoencoder. *CS294A Lecture notes*, 72(2011):1–19, 2011.

[44] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng. Multimodal deep learning. In *ICML*, pages 689–696, 2011.

[45] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. Learning and transferring mid-level image representations using convolutional neural networks. In *CVPR*, pages 1717–1724, 2014.

[46] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–1543, 2014.

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

[48] Scott Reed, Kihyuk Sohn, Yuting Zhang, and Honglak Lee. Learning to disentangle factors of variation with manifold interaction. In *ICML*, pages 1431–1439, 2014.

[49] Salah Rifai, Yoshua Bengio, Aaron Courville, Pascal Vincent, and Mehdi Mirza. Disentangling factors of variation for facial expression recognition. *ECCV*, pages 808–822, 2012.

[50] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[51] Ruslan Salakhutdinov and Geoffrey Hinton. Deep boltzmann machines. In *Artificial Intelligence and Statistics*, pages 448–455, 2009.

[52] Ruhi Sarikaya, Geoffrey E. Hinton, and Anoop Deoras. Application of deep belief networks for natural language understanding. *IEEE/ACM Trans. Audio, Speech & Language Processing*, 22(4):778–784, 2014.

[53] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *CVPR Workshops*, pages 806–813, 2014.

[54] N. Siddharth, Brooks Paige, Jan-Willem van de Meent, Alban Desmaison, Frank Wood, Noah D. Goodman, Pushmeet Kohli, and Philip H. S. Torr. Learning disentangled representations with semi-supervised deep generative models. In *NIPS*, 2017.

[55] Karen Simonyan and Andrew Zisserman. Two-stream convolutional networks for action recognition in videos. In *NIPS*, pages 568–576, 2014.

[56] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[57] Nitish Srivastava and Ruslan R Salakhutdinov. Multimodal learning with deep boltzmann machines. In *NIPS*, pages 2222–2230, 2012.

[58] Nitish Srivastava, Ruslan R Salakhutdinov, and Geoffrey E Hinton. Modeling documents with deep boltzmann machines. In *UAI*, 2013.

[59] Heung-Il Suk, Seong-Whan Lee, Dinggang Shen, Alzheimer's Disease Neuroimaging Initiative, et al. Hierarchical feature representation and multimodal fusion with deep learning for ad/mci diagnosis. *NeuroImage*, 101:569–582, 2014.

[60] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *NIPS*, pages 3104–3112, 2014.

[61] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *WWW*, pages 1067–1077, 2015.

[62] Tijmen Tieleman. Training restricted boltzmann machines using approximations to the likelihood gradient. In *ICML*, pages 1064–1071. ACM, 2008.

[63] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11(Dec):3371–3408, 2010.

[64] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. In *SIGKDD*, pages 1225–1234. ACM, 2016.

[65] Jun Wang, Lantao Yu, Weinan Zhang, Yu Gong, Yinghui Xu, Benyou Wang, Peng Zhang, and Dell Zhang. Irgan: A minimax game for unifying generative and discriminative information retrieval models. In *SIGIR*, 2017.

[66] Suhang Wang, Jiliang Tang, Charu Aggarwal, and Huan Liu. Linked document embedding for classification. In *CIKM*, pages 115–124. ACM, 2016.

[67] Suhang Wang, Jiliang Tang, Fred Morstatter, and Huan Liu. Paired restricted boltzmann machine for linked data. In *CIKM*, pages 1753–1762. ACM, 2016.

[68] Yilin Wang, Suhang Wang, Jiliang Tang, Neil O'Hare, Yi Chang, and Baoxin Li. Hierarchical attention network for action recognition in videos. *CoRR*, abs/1607.06416, 2016.

[69] Max Welling, Michal Rosen-Zvi, and Geoffrey E Hinton. Exponential family harmoniums with an application to information retrieval. In *NIPS*, volume 4, pages 1481–1488, 2004.

[70] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[71] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *ICML*, pages 2048–2057, 2015.

[72] Xinchen Yan, Jimei Yang, Kihyuk Sohn, and Honglak Lee. Attribute2image: Conditional image generation from visual attributes. In *ECCV*, pages 776–791. Springer, 2016.

[73] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alexander J Smola, and Eduard H Hovy. Hierarchical attention networks for document classification. In *HLT-NAACL*, pages 1480–1489, 2016.

[74] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. Seqgan: Sequence generative adversarial nets with policy gradient. In *AAAI*, pages 2852–2858, 2017.

[75] Shengjia Zhao, Jiaming Song, and Stefano Ermon. Learning hierarchical features from deep generative models. In *ICML*, pages 4091–4099, 2017.