AI1001 - Assignment 14

Shivram S
ai24btech11031@iith.ac.in

1 Recurrent Neural Networks

Recurrent neural networks (RNNs) allow cycles in computation graphs. Units may take as input values computed from their own output at an earlier step. This allows the RNN to have internal state or **memory**.

RNNs can be used as tools for analyzing sequential data, similar to hidden Markov models, dynamic Bayesian networks, and Kalman filters. RNNs make a **Markov assumption**: that the hidden state $\mathbf{z}_t$ of the network captures information from all previous inputs. Additionally, the function $f_{\mathbf{w}}$ in the RNN's update process $\mathbf{z}_t = f_{\mathbf{w}}(\mathbf{z}_{t-1}, \mathbf{x}_t)$ must represent a **time-homogeneous** process.

If we used a feedforward network to analyze sequential data, the network could examine only a finite-length window of the data, and the network would fail to detect long-distance dependencies. RNNs address this by keeping track of previous inputs in the hidden state.

1.1 Training an RNN

We will consider a model with an input later $\mathbf{x}$ a hidden layer $\mathbf{z}$ with recurrent connections, and an output layer $\mathbf{y}$. We can define the model using the equations:

$$\begin{aligned}\mathbf{z}_t &= f_{\mathbf{w}}(\mathbf{z}_{t-1}, \mathbf{z}_t) = \mathbf{g}_z(\mathbf{W}_{z,z}\mathbf{z}_{t-1} + \mathbf{W}_{x,z}\mathbf{x}_y) \equiv \mathbf{g}_z(\mathbf{in}_{z,t}) \\ \hat{\mathbf{y}}_t &= \mathbf{g}_y(\mathbf{W}_{z,y}\mathbf{z}_t) \equiv \mathbf{g}_y(\mathbf{in}_{y,t})\end{aligned}$$

Given a sequence of input vectors $\mathbf{x}_1, \ldots, \mathbf{x}_T$ and the observed outputs $\mathbf{y}_1, \ldots, \mathbf{y}_T$, We can "unroll" the RNN into a feedforward network. The weight matrices are shared across all time steps. We can calculate gradients to train the weights using gradient descent, but the sharing of weights makes the computation a little more complicated. For example, to calculate the gradient for the hidden layer weight $w_{z,z}$

$$\begin{aligned}\frac{\partial L}{\partial w_{z,z}} &= \sum_{t=1}^{T} -2(y_t - \hat{y}_t)\frac{\partial \hat{y}_t}{\partial w_{z,z}} \\ &= \sum_{t=1}^{T} -2(y_t - \hat{y}_t)\frac{\partial}{\partial w_{z,z}} g_y(in_{y,t}) \\ &= \sum_{t=1}^{T} -2(y_t - \hat{y}_t) g_y'(in_{y,t}) w_{z,y}\frac{\partial z_t}{\partial w_{z,z}}\end{aligned}$$

We can now calculate the gradient for the hidden unit z_t as

$$\frac{\partial z_t}{\partial w_{z,z}} = \frac{\partial}{\partial w_{z,z}} g_z(in_{z,t}) = g_z'(in_{z,t})\left(z_{t-1} + w_{z,z}\frac{\partial z_{t-1}}{\partial w_{z,z}}\right)$$

Preprint. Under review.

Since the contribution to the gradient from time step t is calculated using the contribution from time step $t-1$, the algorithm runs in linear time and is called **back-propagation through time**.

If $w_{z,z} > 1$, we may experience the **exploding gradient problem**. This can be mitigated using more elaborate RNN design.

1.2 Long short-term memory RNNs

Long short-term memory (LSTM) RNNs are architectures designed to preserve information over many time steps. The long term memory has a **memory cell**, **c**, which is copied from time step to time step. New information enters the memory by adding updates. LSTMs also include **gating units** that control the flow of information.

- The **forget gate f** determines if each element of the memory cell is remembered or forgotten.
- The **input gate i** determines if each element of the memory cell is updated additively by new information from the input vector.
- The **output gate o** determines if each element of the memory cell is transferred to the short-term memory z.

Unlike Boolean functions, gates in LSTM are soft. For example, elements of the memory cell will be partially forgotten if the elements of **f** are small, but not zero. Values of gating units are always in the range $[0, 1]$ as they are obtained as outputs of a sigmoid function. We can write the update equations for the gating units as:

$$\begin{aligned}\mathbf{f}_t &= \sigma(\mathbf{W}_{x,f}\mathbf{x}_t + \mathbf{W}_{z,f}\mathbf{z}_{t-1}) \\ \mathbf{i}_t &= \sigma(\mathbf{W}_{x,i}\mathbf{x}_t + \mathbf{W}_{z,i}\mathbf{z}_{t-1}) \\ \mathbf{o}_t &= \sigma(\mathbf{W}_{x,o}\mathbf{x}_t + \mathbf{W}_{z,o}\mathbf{z}_{t-1})\end{aligned}$$

The update rules for the memory are given by

$$\begin{aligned}\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tanh(\mathbf{W}_{x,c}\mathbf{x}_t + \mathbf{W}_{z,c}\mathbf{z}_{t-1}) \\ \mathbf{z}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)\end{aligned}$$

where $\odot$ denotes elementwise multiplication.

2 Unsupervised Learning

Supervised learning algorithms are given a training set of inputs and corresponding outputs. Unsupervised learning algorithms, on the other hand, take a training set of unlabeled examples **x**. The algorithm might try to learn new representations, or it might learn a generative model from which new samples can be generated.

Suppose we learn a joint model $P_W(\mathbf{x}, \mathbf{z})$ where **z** is a set of latent, unobserved variables that represent the content of **x** in some way. The model can associate **z** with **x** however it chooses. The model can achieve both **representation learning** (constructing meaningful **z** from **x**) and **generative modelling** (determining $P_W(\mathbf{x})$)

2.1 Probabilistic PCA

In a **probabilistic principal component analysis** (PPCA), **z** is chosen from a zero-mean, spherical Gaussian. **x** is generated from **z** by applying a weight matrix **W** and adding spherical Gaussian noise (with noise parameter σ^2). The weights can be learnt by maximizing the likelihood of the data.

$$P_W(\mathbf{x}) = \int P_W(\mathbf{x}, \mathbf{z}) = \mathcal{N}(\mathbf{x}; \mathbf{0}, \mathbf{W}\mathbf{W}^\top + \sigma^2\mathbf{I})$$

This can be done by gradient methods, or by an efficient EM algorithm. Once $\mathbf{W}$ has been learned, new data samples can be generated directly from $P_W(\mathbf{x})$. Additionally, new observations that have very low probability can be flagged as potential anomalies.

The dimensionality of $\mathbf{z}$ is much less than the dimensionality of $\mathbf{x}$, so the model learns to explain the data in terms of a small number of features. These features can be extracted for use in classifiers by computing $\hat{\mathbf{z}}$, the expectation of $P_W(\mathbf{z}|\mathbf{x})$.

We can generate data from a PPCA model by sampling $\mathbf{z}$ from the Gaussian prior, then sampling $\mathbf{x}$ from a Gaussian with mean $\mathbf{Wz}$.

2.2 Autoencoders

An **autoencoder** is a model containing two parts: an encoder that maps from $\mathbf{x}$ to a representation $\hat{\mathbf{z}}$, and a decoder that maps from $\hat{\mathbf{z}}$ to observed data $\mathbf{x}$. The model is trained so that $\mathbf{x} \approx g(f(\mathbf{x}))$, where f is the encoder function, and g is the decoder function. f and g may be simple linear models, or they can be represented by a deep neural network.

The linear autoencoder is a simple autoencoder where:

$$\hat{\mathbf{z}} = f(\mathbf{x}) = \mathbf{W}\mathbf{x}$$
$$\mathbf{x} = g(\hat{\mathbf{z}}) = \mathbf{W}^\top \hat{\mathbf{z}}$$

This model can be trained by minimizing the squared error. The idea is to train $\mathbf{W}$ so that a low-dimensional $\hat{\mathbf{z}}$ will retain as much information as possible to reconstruct $\mathbf{x}$.

When $\mathbf{z}$ is m-dimensional, $\mathbf{W}$ must learn to span the m principal components of the data, which are the m eigenvectors that have the largest eigenvalues. Thus, the linear autoencoder turns out to be closely connected to classical PCA.

The correspondence between classical PCA models and linear autoencoders suggest that there might be a way to capture more complex kinds of generative models using more complex autoencoders. The **variational autoencoder** (VAE) provides one way to do it.The idea is to use a **variational posterior** $Q(\mathbf{z})$ drawn from a computationally tractable family of distributions as an approximation to the true posterior.

Q is optimized to be "as close as possible" to the true distribution $P(\mathbf{z}|\mathbf{x})$. This is done by minimizing KL divergence:

$$D_{KL}(Q(\mathbf{z})\|P(\mathbf{z}|\mathbf{x})) = \int Q(\mathbf{z}) \log \frac{Q(\mathbf{z})}{P(\mathbf{z}|\mathbf{x})} d\mathbf{z}$$

KL divergence is zero when Q and P conincide, and positive otherwise. We can define the **variational lower bound** or **evidence lower bound** (ELBO), $\mathcal{L}$ as

$$\mathcal{L}(\mathbf{x}, Q) = \log P(\mathbf{x}) - D_{KL}(Q(\mathbf{x})\|P(\mathbf{z}|\mathbf{x}))$$

Variational learning maximizes $\mathcal{L}$ in the hope that the solution will be close to maximizing $\log P(\mathbf{x})$. We may rewrite the expression for $\mathcal{L}$ as

$$\mathcal{L} = H(Q) + \mathbf{E}_{\mathbf{z}\sim Q} \log P(\mathbf{z}, \mathbf{x})$$

For some Q (such as Gaussian distributions), $H(Q)$ can be evaluated analytically. $\mathbf{E}_{\mathbf{z}\sim Q} \log P(\mathbf{z}, \mathbf{x})$ can be estimated using samples of $\mathbf{z}$ from Q. $P(\mathbf{z}, \mathbf{x})$ can usually be evaluated efficiently: for example if P is a Bayes net, then $P(\mathbf{z}, \mathbf{x})$ is just a product of conditional probabilities.

2.3 Deep autoregressive models

An **autoregressive model** is one in which each element of the data vector is predicted based on other elements of the vector. If $\mathbf{x}$ is of fixed size, an AR model can be thought of as a fully observable and

possibly fully connected Bayes net. This makes it easy to calculate the likelihood of a given data vector, and to predict the value of a simple missing variable, given all others.

AR models are commonly used in the analysis of time series data, where an AR model of order k predicts x_t given $x_{t-k}, \ldots, x_{t-1}$. Hence, an n-gram model is an AR model of order $n-1$.

In classical AR models, the conditional distribution $P(x_t|x_{t-k}, \ldots, x_{t-1})$ is a linear Gaussian model with a fixed variance whose mean is a weighted linear combination of $x_{t-k}, \ldots, x_{t-1}$, i.e., a standard linear regression model. The maximum likelihood is given by the **Yule-Walker** equations.

A **deep autoregressive model** is one in which the linear Gaussian model is replaced by an arbitrary deep network with a suitable output layer. Recent applications of deep autoregressive models include DeepMind'S WaveNet speech generation model, which implements a nonlinear AR model of order 4800 with a convolutional structure.

2.4 Generative adversarial networks

A **generative adversarial network** (GAN) is a pair of networks: a **generator** which maps values from $\mathbf{z}$ to $\mathbf{x}$ to produce samples from $P_W(\mathbf{x})$, and a **discriminator**, which is a classifier trained to classify $\mathbf{x}$ as real (from the training set), or fake (created by the generator). GANs are **implicit models**, in that samples can be generated, but probabilities are not available.

The generator of a GAN is closely related to the decoder of a VAE. Both the generator and the discriminator are trained simultaneously. The competition between generator and discriminator can be described using game theory. The idea is that in the equilibrium state, the generator should reproduce the training distribution perfectly, so that the discriminator can't perform better than random guessing.

GANs have worked particularly well for image-generation tasks.

2.5 Unsupervised translation

Translation tasks involves multidimensional data such as images and natural language which have statistical dependence between the various dimensions. Such data is said to have "rich structure." Translation tasks consist of transforming an input $\mathbf{x}$ that has rich structure to an output $\mathbf{y}$ that also has rich structure.

In supervised translation, the data consists of many $(\mathbf{x}, \mathbf{y})$ pairs, and the model maps each $\mathbf{x}$ to the corresponding $\mathbf{y}$. Unsupervised translation trains on many $\mathbf{x}$ and separate $\mathbf{y}$, but no corresponding $(\mathbf{x}, \mathbf{y})$ pairs.

Most unsupervised translation approaches are based on GANs. The GAN training framework makes it possible to train a generator that generates any of the possible samples that the the discriminator accepts as a realistic example of $\mathbf{y}$ given $\mathbf{x}$, without any need for a specific paired $\mathbf{y}$.

2.6 Transfer learning and multitask learning

In **transfer learning**, experience with one learned task helps an agent learn better on another task. We may take a network trained for a task A, and update its weights using gradient descent for task B. We may use a smaller learning rate in task B, depending on the similarity of the tasks, and how much data was used in task A.

There are many high-quality pre-trained models available, which has contributed to the popularity of transfer learning. For example, using a pre-trained visual object TODO: AR models recognition model such as ResNet-50 can help with identification of relevant features, thus saving weeks of training time.

If the tasks are very similar, we might also freeze the first few layers of the model as they serve as feature detectors that will be useful for the new model. The later layers are the ones that identify problem-specific features.

In natural language processing, models such as RoBERTa are pre-trained on the vocabulary and syntax of everyday language. Such models can be fine-tuned with domain-specific vocabulary, and

trained on task-specific data. For example, a model for answering questions might be trained on question/answer pairs.

The controller for a self-driving car might be trained on a simulator, but in the real world, the environment can vary greatly. The model can be fine-tuned with real-time data from an actual vehicle, and thus it can adapt quickly to the new environment.

Multitask learning is a form of transfer learning in which we simultaneously train a model on multiple objectives. This allows the model to create a "common representation" that reflects similarities between the tasks. For example, a natural language model might be trained simultaneously on part-of-speech tagging, document classification, language detection, word prediction, etc.

3 Applications

3.1 Vision

Computer vision is the application area which had the biggest impact on deep learning. Deep convolutional networks have been used in handwriting recognition, speech generation, etc.

Deep learning was popularized by the AlexNet image classification system, which managed to achieve an error rate of 15.3% on the ImageNet competition. The model had five convolutional layers, interspersed with max-pooling layers, followed by three fully connected layers. It took advantage of GPUs to speed up the training process.

Since 2012, the top-5 error rate on ImageNet has been reduced to less than 2%. CNNs have been applied in a wide range on visual tasks. Self driving is among the most demanding of visual tasks.

3.2 Natural language processing

Deep learning has had a huge impact on tasks such as translation and speech recognition. There is the possibility of end-to-end learning and automatic generation of representations for the meanings of words.

In translation tasks, the classical pipeline approach, which corresponds to how a human translator works, is outperfoemd by end-to-end methods. Machine translation systems are approaching human performance for languages pairs such as French and English, which have large paired data sets available.

There is some evidence that networks trained on multiple languages lean an internal meaning representations. For example, learning Portuguese/English and English/Spanish translations have allowed models to perform Portuguese/Spanish translations without any Portuguese/Spanish training pairs.

The representation of words as vectors in a high-dimensional space, known as **word embeddings**, has shown promise. Because words with similar meanings are used in similar contexts, they end up near each other in the vector space. This allows the network to generalize across categories of words.

3.3 Reinforcement learning

In reinforcement learning (RL), an agent learns from a series of reward signals that provide some indication to the quality of its behavior. The goal is to optimize the sum of future rewards. The agent can learn a value function, a Q-function, a policy, and so on.

While the methods of training in RL differ from those of supervised learning, the ability of multilayer computation graphs to represent complex functions over large input spaces has lead to the development of the field of **deep reinforcement learning**.

The first major demonstration of deep RL was DeepMind's Atari playing agent, DQN. The agent learnt a Q-function from raw image data, with the reward signal begin the game score. DeepMind's AlphaGo system also used deep RL to defeat the best human players at the game of Go.

Despite its successes, deep RL still faces significant obstacles. It is often difficult to get good performance, and trained systems may behave very unpredictably if the environment differs even a little from training data.