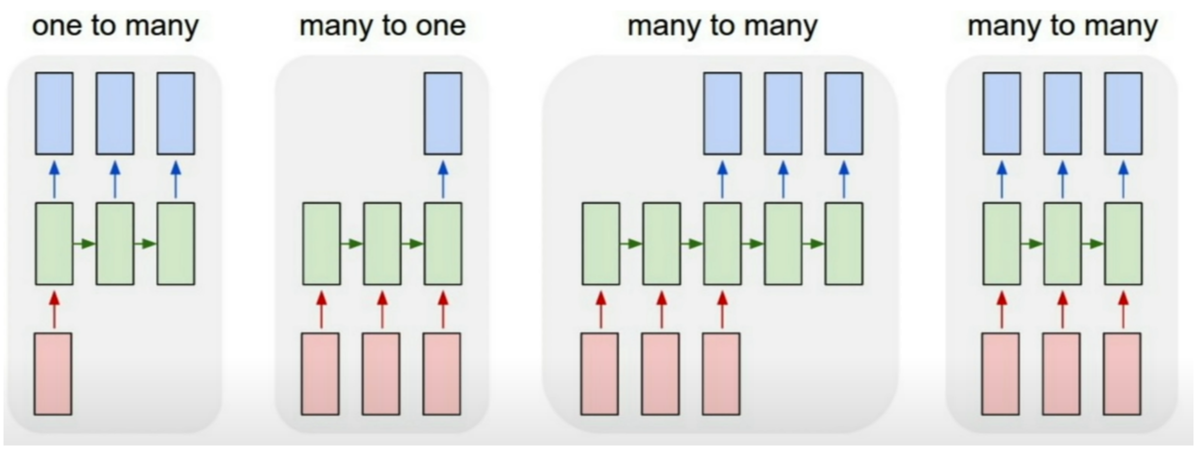
Recurrent Neural Networks

Process sequences

For vanilla neural network, we receive some input and produce a single output. In some context of machine learning, we want to have more flexibility in the types of data that out models can process.

So once we move to recurrent neural networks, we can play around with the types of input and output data. This is what we can do if we are capable of processing sequential data.

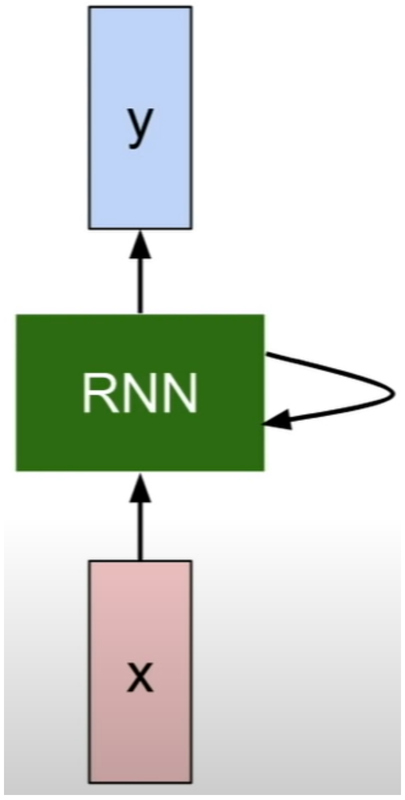


one to many	**Image Captioning (image -> sequence of words)**
many to one	**Sentiment Classification (sequence of words -> sentiment)**
many to many	**Machine translation (words -> words)**

Intuition

In general, RNN has a little recurrent core cell and it will take some input x. Moreover, RNN has some internal hidden state and the internal hidden state will be updated every time RNN reads a new input.

And frequently, we will want out RNN to also produce some output at every time step. So we will have this pattern: read an input -> update the hidden state -> produce and output.



In side the little green RNN block, we are computing some recurrent relation with a function f:

$$h_t = f_W(h_{t-1}, x_t)$$

where at time step t, we change the old state h_{t-1} into the new state h_t, with the input vector x_t. Note that the same function and the same set of parameters are used every time step.

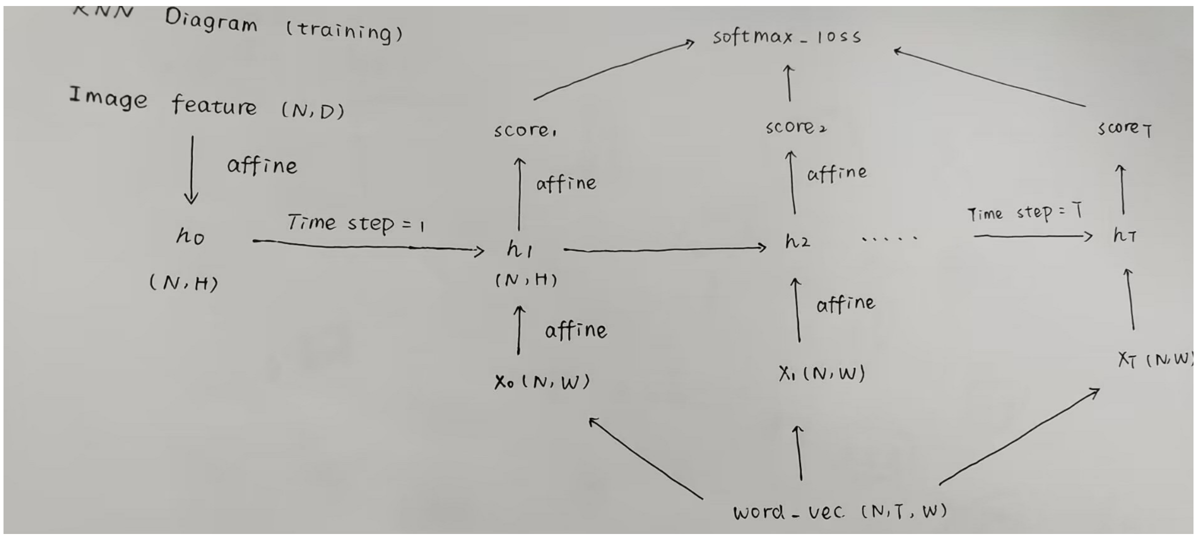
Vanilla RNN

Then kind of the simple function form you can imagine is what we call this vanilla recurrent neural network. We have some weight matrix W_{hh} and W_{xh} and do matrix multiplication against the two state :

$$h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

Let's take the image captioning with RNN as a running example to get a deeper understanding. Check `\assignment3\RNN_Captioning.ipynb` to understand the background better.



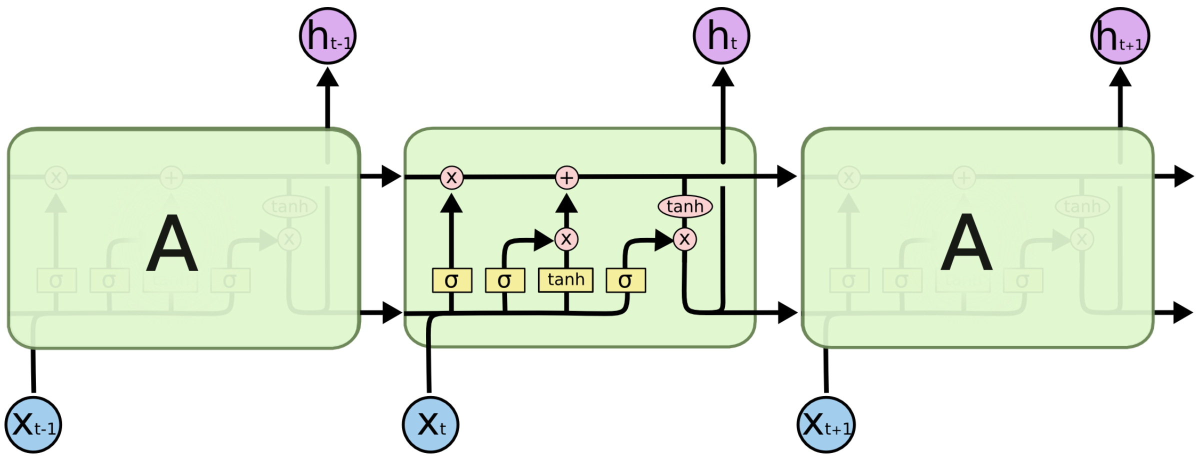
In the training process, we transform the image feature into the initial state h_0 and feed the ground-truth caption into the neural network. While testing, we compute the word with the highest score in time step $t-1$ and feed it into the time step t.

LSTM

Long Short Term Memory networks - usually just called "LSTMs" - are a special kind of RNN, capable of learning long-term dependencies.

All recurrent neural networks have the form of a chain of repeating modules of neural network. In vanilla RNNs, this repeating module have a very simple structure, such as a single tanh layer.

LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.



The key to LSTMs is **cell state**, the horizontal line running through the diagram. Just like items on the conveyor belt, it's easy for information to just flow along it unchanged.

The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called **gates**.

Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation. Since sigmoid outputs a value in $[0, 1]$, 0 represents "completely get rid of this" while 1 represents "completely keep this".

There are overall four kinds of gates:

Name	Equation	Usage
Forget gate	$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$	Keep or forget c_{t-1}
Gate gate	$g_t = tanh(W_g[h_{t-1}, x_t] + b_g)$	Add the new information to c_t
Input gate	$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$	Filter the information in Gate gate
Output gate	$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$	Decide what to output

Through gates, we'd actually drop the information about the old subject and add the new information (recall **soft update**) :

$$c_t = c_{t-1} \odot f + i \odot g$$

where $\odot$ is the elementwise product of vectors. We compute hidden state by selecting what we want and tanh is applied to keep the value between $[-1, 1]$:

$$h_t = tanh(o \odot c_t)$$

In the code, we assume that data is stored in batches so that $X_t \in \mathbb{R}^{N \times D}$ and will work with parameters: $W_x \in \mathbb{R}^{D \times 4H}$, $W_h \in \mathbb{R}^{H \times 4H}$ so that activations $A \in \mathbb{R}^{N \times 4H}$ can be computed efficiently as:

$$A = X_t W_x + H_{t-1} W_h$$

Reference

Generative Adversarial Nets

The adversarial modeling framework is most straightforward to apply when the models are both multiplayer perceptrons. In a GAN, we build two different neural networks.

Our first network is a traditional classification network, called **discriminator**. The discriminator takes images as input and classify them as real (training data) or fake (generated images).

The Second network, called **generator**, will take a random noise as the input and produce picture. The goal of the generator is to fool the discriminator into thinking the images it produced are real.

The training process can be viewed as two players who want to defeat each other. And the game has a score:

$$\mathbb{E}_{x\sim p_{data}}[\log D(x)] + \mathbb{E}_{z\sim p(z)}[\log(1 - D(G(z)))]$$

> More specific about the expression $\mathbb{E}_{x\sim p_{data}}$. This means that we sample x from the distribution p_{data} (we consider the training data as a **distribution** of pictures). In other words:
>
> $$\mathbb{E}_{x\sim p_{data}}[\log D(x)] = \int_x p_{data}(x) \cdot \log D(x) \cdot dx$$

The score comes from the cross entropy in logistic regression. The discriminator(D) tries to maximize the score while the generator(G) wants to minimize it:

$$\min_G \max_D \mathbb{E}_{x\sim p_{data}}[\log D(x)] + \mathbb{E}_{z\sim p(z)}[\log(1 - D(G(z)))]$$

In gradient ascent, we will alternate the following updates:

1. For generator: maximize the probability of the discriminator making incorrect choice on generated data:

   $\text{maximize}_G \ \mathbb{E}_{z\sim p(z)} \log D(G(Z))$

2. For discriminator: maximize the probability of the discriminator making correct choice on data:

   $\text{maximize}_D \ \mathbb{E}_{x\sim p_{data}}[\log D(x)] + \mathbb{E}_{z\sim p(z)}[\log(1-D(G(z)))]$

You might be confused about why we maximize the probability of wrong answers instead of minimizing the probability of correct answers. That is because early in the learning, when G is poor, D can easily distinguish them from the training data. In this case, $\log(1-D(G(z)))$ saturates, which make it hard to apply gradient descent. On the contrast, maximizing $\log D(G(z))$ can provide much stronger gradient.

Theoretical Results

We will show the minimax game has a global optimum of $p_g = p_{data}$. Firstly, we consider the optimal discriminator D for a given generator G.

Proposition 1. For G fixed, the optimal D is:

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

Proof. The discriminator wants to maximize:

$$\begin{aligned}
&\int_x p_{data}(x) \cdot \log D(x) \cdot dx + \int_z p_z(z) \cdot \log(1 - G(D(z))) \cdot dz \\
&= \int_x p_{data}(x) \cdot \log D(x) \cdot dx + \int_x p_g(x) \cdot \log(1 - D(x)) \cdot dx \\
&= \int_x p_{data}(x) \cdot \log D(x) + p_g(x) \cdot \log(1 - D(x)) dx \\
&\leq \int_x p_{data}(x) \cdot \log \frac{p_{data}(x)}{p_{data}(x) + p_g(x)} + p_g(x) \cdot \log \frac{p_g(x)}{p_{data}(x) + p_g(x)} dx
\end{aligned}$$

If and only if $D(x) = \frac{p_{data}(x)}{p_{data}(x)+p_g(x)}$ holds, the maximum can be achieved.

Theorem 1. The global minimum is achieved if and only if $p_{data} = p_g$.

Proof. The generator tries to minimize:

$$\begin{aligned}
&\mathbb{E}_{x\sim p_{data}} \log \frac{p_{data}(x)}{p_{data}(x) + p_g(x)} + \mathbb{E}_{x\sim p_g} \log \frac{p_g(x)}{p_{data}(x) + p_g(x)} \\
&= -\log 4 + \mathbb{E}_{x\sim p_{data}} \log \frac{p_{data}(x)}{\frac{p_{data}(x)+p_g(x)}{2}} + \mathbb{E}_{x\sim p_g} \log \frac{p_g(x)}{\frac{p_{data}(x)+p_g(x)}{2}} \\
&= -\log 4 + KL(p_{data}||\frac{p_{data}(x) + p_g(x)}{2}) + KL(p_g||\frac{p_{data}(x) + p_g(x)}{2}) \\
&\geq -\log 4
\end{aligned}$$

> where KL means Kullback-Leibler Divergence:
>
> $KL(P||Q) = \sum_i P(i) \cdot \log \frac{P(i)}{Q(i)}$
>
> $KL(P||Q) \geq 0$, $KL(P||Q) = 0$ if and only if $P = Q$ holds:
>
> $\sum_i P(i) \cdot \log \frac{Q(i)}{P(i)} \leq \sum_i P(i) \cdot (\frac{Q(i)}{P(i)} - 1) = \sum_i (Q(i) - P(i)) = 0$
>
> Therefore:

> $KL(P||Q) = \sum_i P(i) \cdot \log \frac{P(i)}{Q(i)} \geq 0$

So the global minimum achieve when the KL-divergence equal to 0. At this point $p_{data} = p_g$.

The theoretical result shows that the generator will perfectly fit into the training data after sufficient iterations. This can also be viewed as a Nash equilibrium between the two players. No player can improve their own outcome by unilaterally changing their strategy. In other words, this is an inevitable consequence within the context of game theory.

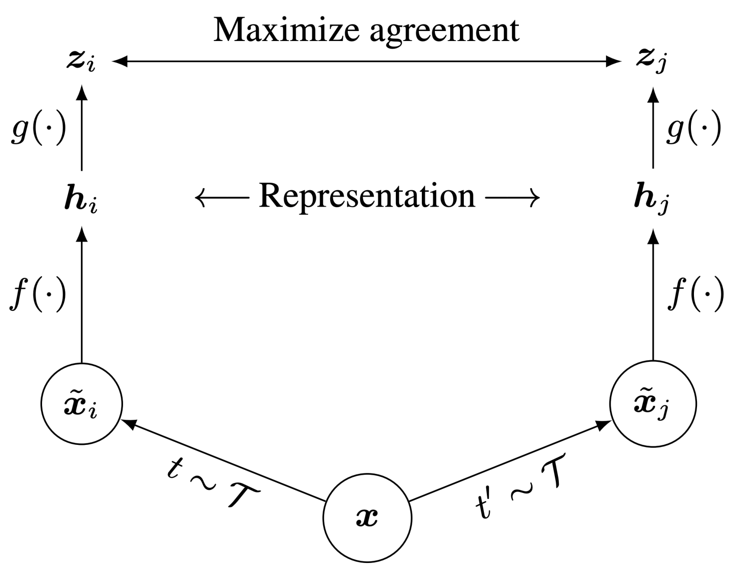
Contrastive Learning

Modern day machine learning needs lots of labeled data. But often times it's hard to collect large amount of human-labeled data. Is there a way we could ask machines to automatically learn a model which can generate good visual representations without a labeled dataset? Yes, enter self-supervised learning!

A good representation vectors capture the important features of the images as it relates to the rest of the data. This means that images representing the similar entities should have similar representation vectors, while different images should have different entities.

Recently, SimCLR introduces a new architecture which uses **contrastive learning** to learn good visual representations. Specifically, for each image in the dataset, SimCLR generates two differently augmented views of that image, called a **positive pair**. Then, the model is encouraged to generate similar representation vectors for this pair of images.

Here is the architecture of the contrastive learning:



$\tilde{x}_i$ and $\tilde{x}_j$ are the augmented data that we generated from the same picture x. Then, we pass $\tilde{x}_i$ and $\tilde{x}_j$ into a Resnet f. Finally, a small neural network maps the representation to the space where the **contrastive loss** is applied. The contrastive loss is defined as follow:

$L = \frac{1}{2N} \sum_{i=1}^{N} [l(i, i+N) + l(i+N, i)]$

where $l(i, j)$ is the normalized temperature-scaled cross entropy loss and aims to maximize the agreement of z_i and z_j relative to all other augmented examples in the batch:

$$l(i, j) = \frac{\exp(\cos(z_i, z_j)/\tau)}{\sum_{k \neq i} \exp(\cos(z_k, z_j)/\tau)}$$

where $\cos(z_i, z_j)$ means the similarity of vectors z_i and z_j, and τ represents the temperature.