Jianmging's Notes

Contents

1 Sampling

Monte Carlo Integration is used to compute the sum of the value $f(x)$ over x in complex structures. [1]

$$\int f(x)dx \tag{1}$$

It is hard to directly compute 1 if $f(x)$ is complicated. We use sampling to estimate

$$E[f(x)] = \frac{1}{N}\sum_i f(x_i) \quad x_i \; w.r.t. \; p(x) \tag{2}$$

where x has a value from $f(x)$ and x appear with the probability $p(x)$

1.1 Direct Sampling (Inverse Sampling)

We assume that the x appear uniformly, and the cumulative probability $p(x)$ is $h(x)$. 1) get y from a uniform and obtain $x = h^{-1}(y)$ where the function h^{-1} is the inverse of the function h. Then we can compute 2.

There are two reasons why we use the cumulative probability and inverse function, the first reason is (straightforward) the value range of h^{-1} is [0,1] that equals to the range of uniform and x to y is bidirectional projection, and there is a proof for second reason.

If the inverse function is not easily computed, we have to look for other sampling methods.

1.2 Rejection Sampling

If x is hard to sample from $p(x)$, we sample x from another distribution $q(x)$ that is simple and easy to sample. Sometime $q(x)$ is uniform or Gaussian. We have to design a constant k to make sure all the value from $kq(x)$ is larger or equal to the value from $p(x)$ (i.e., upper bound).

We sample x from $q(x)$ and randomly pick u from uniform, if $u \leq \frac{p(x)}{kq(x)}$, the x is accepted. The intuition is that we pick the x which appear frequent and ignore the low-frequency x, and high-frequency x contributes more to estimation.

1.3 Importance Sampling

Similar to the rejection sampling, importance sampling samples x from $q(x)$ instead of directly from $p(x)$. The difference is that rejection sampling use uniform to choose which one is accepted while importance sampling accept all with weights that are identified by $p(x)/q(x)$.

Both for rejection and importance sampling, $p(x)$ is not necessary to be normalized, usually we use $\bar{p}(x)$ instead of $p(x)$. An example is to compute gradience on softmax over the whole vocabulary in language models.

[1] `https://www.jianshu.com/p/3d30070932a8`,`https://yq.aliyun.com/articles/627960`

1.4 Markov Chain Mento Carlo (MCMC) sampling

In sampling, what we want is that all x we sampled should satisfy $p(x)$. However, we hardly sample x from $p(x)$, so the method above sample from a easy-and-similar distribution $q(x)$. In MCMC, we simulate a Markov chain that can use states ($\pi(i)$) and transition (P_{ij}) to satify $p(x)$, only if MC is converged. If MC is converged, we can sample from the states distribution $\pi(i)$, and next, we keep the current state $\pi(i)$, $\pi(j)$ or according to the P_{ij}, change to another state to sample. [2]

However, it is slow to converged, and then **Metropolis-Hastings** method is proposed, which normalizes accept rate. However, the accept rate is expensitive to compute on large dimension. **Gibbs** adopts conditions probability from different features (dimensions).[3]

2 Variational Auto-Encoder (VAE) vs Expectation Maximization (EM)

For the observed variable x and latent variable z, $p_\theta(x) = \int_z p_\theta(x,z)$.

If we want to estimate the parameters θ, we can do Maximum Likelihood Estimation (MLE) or Maximum-A-Posterior (MAP) estimation.[4]

2.1 Expectation-Maximization Algorithm (EM) for MLE

E step fixes θ and compute the distribution of $p_\theta(z|x)$. 比如，聚类中隐变量z是数据点属于某个类的概率。当固定θ时，我们可以得到每个样本属于某个类的概率。再比如，词对齐中源语言端某个词对应到目标端的某个词的概率为隐变量z。当固定θ时，我们可以得到词对齐概率

M step maximizes likelihood function, because we have outputs of E steps, similar to supervised learning with the outputs of E steps. 我们有了E步因变量的概率，就可以做sample生成大量带隐变量的样本，做类似于类监督的学习，更新θ

However, EM algorithm is broken if $p_\theta(z|x)$ is intractable. In E steps, we cannot get the distribution over z, so we cannot update θ in M steps. [5]

2.2 Mean-Field Variational Inference for MAP

Sometimes we can use a another distribution family $q_\phi(z)$ to approximate the posterior $p_\theta(z|x)$.

$$\prod_{i=1}^{N} q_{\phi_i}(z) \approx \prod_{i=1}^{N} p_\theta(z|x) \tag{3}$$

[2] https://zhuanlan.zhihu.com/p/30003899,https://www.cnblogs.com/xbinworld/p/4266146.html

[3] https://applenob.github.io/machine_learning/MCMC/

[4] https://maurocamaraescudero.netlify.app/post/variational-auto-encoders-and-the-expectation-maximizatio

[5] http://sofasofa.io/tutorials/gmm_em/

where each data point has an independent ϕ_i. if they share the same ϕ across the whole data point, it is called amortized inference.

2.3 VAE

VAE are auto-encoding variational bayes (AEVB) where the probability distributions in the latent variable models are parametrized by Neural Networks, and they share the paramters for each data pointer (amortized inference)

$q_\phi(z|x)$ is used to approximate true posterior distribution $p_\theta(z|x)$. Our purpose is to minimize the difference between $q_\phi(z|x)$ and $p_\theta(z|x)$, so we consider KL-divergence

$$
\begin{aligned}
& KL(q_\phi(z|x)||p_\theta(z|x)) \\
&= \mathbb{E}_{q_\phi}[\log q_\phi(z|x) - \log p_\theta(z|x)] && \text{KL definition} \\
&= \mathbb{E}_{q_\phi}[\log q_\phi(z|x) - \log p_\theta(x,z) + \log p_\theta(x,z) - \log p_\theta(z|x)] \\
&= \mathbb{E}_{q_\phi}[-\log \tfrac{p_\theta(x,z)}{q_\phi(z|x)} + (\log \tfrac{p_\theta(x,z)}{p_\theta(z|x)})] \\
&= \mathbb{E}_{q_\phi}[-\log \tfrac{p_\theta(x,z)}{q_\phi(z|x)} + \log p_\theta(x)] && \text{Bayes} \\
&= -\mathbb{E}_{q_\phi}[\log \tfrac{p_\theta(x,z)}{q_\phi(z|x)}] + \log p_\theta(x) && \text{Take } p \text{ out of the expectation on } q \\
&:= -L_{\theta,\phi}(x) + \log p_\theta(x)
\end{aligned}
\tag{4}
$$

$L_{\theta,\phi}(x)$ is evidence lower bound (ELBO). If we maximize ELBO, we will minimize KL which means that q will be closer to p, and also we will maximize $\log p_\theta(x)$ because KL is always equal or larger than 0.

Meanwhile, from Eq:4, the ELBO can be rewrite as

$$
\begin{aligned}
& L_{\theta,\phi}(x) \\
&= \log p_\theta(x) - KL(q_\phi(z|x)||p_\theta(z|x)) \\
&= \log p_\theta(x) - \mathbb{E}_{q_\phi}[\log q_\phi(z|x) - \log p_\theta(z|x)] \\
&= \mathbb{E}_{q_\phi}[\log p_\theta(x) - \log q_\phi(z|x) + \log p_\theta(z|x)] \\
&= \mathbb{E}_{q_\phi}[\log p_\theta(x) + \log p_\theta(z|x) - \log q_\phi(z|x)] \\
&= \mathbb{E}_{q_\phi}[\log p_\theta(x) p_\theta(z|x) - \log q_\phi(z|x)] \\
&= \mathbb{E}_{q_\phi}[\log p_\theta(x|z) p_\theta(z) - \log q_\phi(z|x)] && \text{Bayes} \\
&= \mathbb{E}_{q_\phi}[\log p_\theta(x|z) + \log p_\theta(z) - \log q_\phi(z|x)] \\
&= \mathbb{E}_{q_\phi}[\log p_\theta(x|z)] - \mathbb{E}_{q_\phi}[\log q_\phi(z|x) - \log p_\theta(z)] \\
&= \mathbb{E}_{q_\phi}[\log p_\theta(x|z)] - KL(q_\phi(z|x)||p_\theta(z))
\end{aligned}
\tag{5}
$$

This is the sum of two terms: expected reconstruction error and the KL divergence between the approximation and the latent prior.

For ELBO optimization, ELBO is not differential on ϕ, because there is expectation on q_ϕ. We use the reparamterization trick, which rewrite the sample $z \sim q_\phi(z|x)$ as a ***deterministic*** function of x and ε, parameterized by ϕ, i.e. $z = g_\phi(x,\varepsilon),\ \varepsilon \in N(0,1)$.

3 Generative Adversarial Network for NLP

Given random noisy (vector), an generator G outputs texts, while given texts, an discriminator D judges the texts. Actually both generators and discriminators can do generation (**only positive examples are given**). Generators do bottom-up generation, e.g., generators build the text sequentially. Discriminator do top-down generation, e.g., sampling the negative examples to train the model and higher the scores of positive examples and lower the scores of negative examples.

The motivation of GAN: in image recognition, image 1 can be written in different ways, but it is still image of number 1. If we only use generator, the generators will average the pixel scores both different 1s. If we only use discriminator, the sampling is the problem and sometimes $argmax_{x \in X}$ has intractable X.

The basic idea (goal) of GAN: G generates the negative examples to fool D, and D is to be smart to judge the generated and true examples. Algorithm 1 is the basic training method for GAN. where sometimes $m = n$. **Noted that** steps 1-6 train the discriminator, and steps

Algorithm 1 the train on GAN

1: sample noisy $z_1, ..., z_m$ from $p(z)$
2: given $z_1, ..., z_m$, G generate the negative examples $\bar{x}_1, ..., \bar{x}_m$
3: sample positive examples $x_1, ..., x_n$ from observed data.
4: fix G and update D to maximize V
5: $V = \frac{1}{n}\sum_{i=1}^{n} \log D(x_i) + \frac{1}{m}\sum_{i=1}^{m} \log(1 - D(\bar{x}_i))$
6: $\theta_d \leftarrow \theta_d + \lambda \nabla V$
7: sample noisy $z_1, ..., z_m$ from $p(z)$
8: given $z_1, ..., z_m$, G generate the negative examples $\bar{x}_1, ..., \bar{x}_m$
9: fix D and update G to maximize V
10: $V = \frac{1}{m}\sum_{i=1}^{m} \log(D(\bar{x}_i))$
11: $\theta_g \leftarrow \theta_g + \lambda \nabla V_g$

7-11 train the generator. Sometimes we should train multiple steps on the discriminator before train the generator, which purpose is to find the JS divergence between p_g and p_{data}. Sometimes we should just slightly change the generator, which purpose is to ensure the JS divergence will be smaller and converged. In lines 9-11, fix D and update G to minimize V which is the same to line 5, but the first term has no G, so minimize $V = \frac{1}{m}\sum_{i=1}^{m} \log(1 - D(\bar{x}_i))$ (Minmax GAN) that equals to maximize $V = \frac{1}{m}\sum_{i=1}^{m} \log(D(\bar{x}_i))$(Non-saturating GAN).

3.1 Conditional GAN

Sometimes, we have to generate the text given the constraints (e.g., sentiments and other labels, semantics and other descriptions). The input of G are the noisy z and the conditions c, and the output is the text. The input of D are the generated text $\bar{x}$, the conditions c and

the true text x, and the output is judgement. The positive judgement is that text is true and conditions are paired to the text. Algorithm 2 is the training method for condition GAN

Algorithm 2 the train on condition GAN

sample noisy $z_1, ..., z_m$ from $p(z)$
sample condition $c_1, ..., c_m$ from the observed data
given $z_1, ..., z_m$ and $c_1, ..., c_m$, G generate the negative examples $\bar{x}_1, ..., \bar{x}_m$
take positive examples $x_1, ..., x_m$ from observed data according to the sampled condition $c_1, ..., c_m$
sample $\hat{x}_1, ..., \hat{x}_m$ from the observed data, w.r.t $\hat{x}_i$ and c_i ($i \in [1, m]$) are not paired
fix G and update D
    $V_d = \frac{1}{n}\sum_{i=1}^{n} \log D(c_i, x_i) + \frac{1}{m}\sum_{i=1}^{m} \log(1 - D(c_i, \bar{x}_i)) + \frac{1}{m}\sum_{i=1}^{m} \log(1 - D(c_i, \hat{x}_i))$
    $\theta_d \leftarrow \theta_d + \lambda \nabla V_d$
sample $z_1, ..., z_m$ from $p(z)$
sample condition $c_1, ..., c_m$ from the observed data
given $z_1, ..., z_m$ and $c_1, ..., c_m$, G generate the negative examples $\bar{x}_1, ..., \bar{x}_m$
fix D and update G
    $V_g = \frac{1}{m}\sum_{i=1}^{m} \log(D(c_i, \bar{x}_i))$
    $\theta_g \leftarrow \theta_g + \lambda \nabla V_g$

Text-to-image application: **Stack GAN** generates the small image (64x64) and then generates the large image (128x128), which is similar to coarse-to-fine generation. Image-to-Image application: **Patch GAN** judges the input on local (patch) images instead of the whole image. These variants make the training easier.

However, conditional GAN always requires the pairs of conditions and texts. If we don't know which conditions responds to the texts, we have to do unsupervised conditional learning on GAN

3.2 Unsupervised Conditional GAN

For example, in the task of the style transfer, we have set of text in one style and set of text in anther style, but we don't know alighment between text in two styles (i.e., which two texts are related).

3.2.1 Direct Transformation

The input of the generator G is the text in style A, and the output is the text in style B. The input of the discriminator D is the generated text in style B and the true random text in style B, and the D judges which one is generated. The problem is that the generator will generate the style B but which totally different from the original input.

Ignore Ignore the problem, just train the model. It can work. One possible reason is that the generator will not largely change the input. (Tomer Galanti, ICLR, 2018)

Consistency on Input and Output Add another loss value to make sure the input text is closed to the output text in generators. For example, we can use another pretrained encoder to encode the input and outputs, and make their hidden representation closer. (Yaniv Taigman, ICLR, 2017)

Cycle Consistency (CycleGAN) Add another generator $\bar{D}$ which take the output of the generator D to reconstruct the input, ensuring the reconstructed text is closer to the original input text (cycle is $A \to B \to A$). So $D : A \to B$ and $\bar{D} : B \to A$. In addition, we can also do another cycle $B \to A \leftarrow B$). But in the additional cycle we should use another discriminator D to judge style A. (Jun-Yan Zhu, ICCV, 2017) An arguments: cycle consistency will not ensure the output of the generator can keep the key feature of original input and it will still output a really different text. (Casey Chu, NIPS, 2017) Other cycle consistency GAN: DiscoGAN, DualGAN.

Multiple Style (StarGAN) similar to CycleGAN, but it adds a label as a control to show the style (e.g., 0100101).

3.2.2 Projection to Common Space

Generator will be divided to two parts, a encoder encodes the input to vectors and a decoder reconstruct the input according to the vectors. For style A and B, we have enc_A and dec_A, and enc_B and dec_B. In addition, we have discriminator D_A and D_B to judge the text is generated or not for style A and B. So we have reconstruction error and discriminator error for style A and B. However, the style A and the style B is independent. How to make connection between them.

Structure Sharing We can share the parts of enc_A and enc_B, and share the parts of dec_A and dec_B, or we can add a metric loss to make these parts more similar (not totally share).

Hidden Space discriminator We can add another discriminator to judge the hidden representation comes from style A or style B.

Cycle Consistency $A \to enc_A \to h_A \to dec_B \to B \to enc_B \to h_B \to dec_A \to A$. Similar thing can be taken on style B.

Semantic Consistency $A \to enc_A \to h_A \to dec_B \to B \to enc_B \to h_B$, and then make h_A is closer to h_B. Similar thing can be taken on style B. DTN (Yaniv Taigman, ICLR, 2017) and XGAN (Amelie Royer, arxiv, 2017)

3.3 Generalization on GAN

The original GAN is proposed by minimizing the KL divergence between generated data distribution and true data distribution. In fact, KL divergence can be replaced with other divergence metrics.

3.3.1 f-Divergence

Given two distribution q and p, f-divergence is defined as:

$$D_f(p||q) = \int_x q(x) f(\frac{p(x)}{q(x)}) dx \tag{6}$$

where $f(x)$ is any convex function and $f(1) = 0$. For every convex function $f(x)$, there is conjugate function:

$$f^*(t) = max_{x \in dom f}\{xt - f(x)\} \tag{7}$$

$f^*(t)$ is also a convex function (proof) and $f(x)$ is the conjugate function of $f^*(t)$. If $f(x) = x\log x$, $f^*(t) = exp(t-1)$.

3.3.2 f-GAN

f-GAN aims to minimize the f-divergence. The equation 6 can be rewritten as:

$$\begin{aligned} D_f(p||q) &= \int_x q(x)\ f(\tfrac{p(x)}{q(x)}) dx \\ &= \int_x q(x)\ (\max_{t \in dom(f^*)}\{\tfrac{p(x)}{q(x)} t - f^*(t)\}) dx \\ &\geq \int_x q(x)(\tfrac{p(x)}{q(x)} D(x) - f^*(D(x))) dx \\ &= \int_x p(x) D(x) - \int_x q(x) f^*(D(x)) dx \\ &= \mathbb{E}_{x \sim p(x)}[D(x)] - \mathbb{E}_{x \sim q(x)}[f^*(D(x))] \end{aligned} \tag{8}$$

where $D(x)$ is a function taking a scalar and outputting another scalar. If the D function is optimal that outputs optimal t given x, the $=$ is satisfied. In GAN, $p(x)$ is regarded as the true distribution and $q(x)$ is regarded as the generated distribution.

$$D_f(p_{data}||p_G) \approx \max_D \mathbb{E}_{x \sim p_{data}}[D(x)] - \mathbb{E}_{x \sim p_G}[f^*(D(x))] \tag{9}$$

$$\begin{aligned} G^* &= arg\ \min_G D_f(p_{data}||p_G) \\ &= arg\ \min_G \max_D \mathbb{E}_{x \sim p_{data}}[D(x)] - \mathbb{E}_{x \sim p_G}[f^*(D(x))] \\ &= arg\ \min_G \max_D V(G,D) \end{aligned} \tag{10}$$

There are many divergence function, so there will be many variants of GAN.

3.4 Alternatives to f-GAN

3.4.1 Least Square GAN (LSGAN)

We take the original GAN as an example of f-GAN. The value function is $V = \frac{1}{n}\sum_{i=1}^{n} \log D(x_i) + \frac{1}{m}\sum_{i=1}^{m} \log(1 - D(\bar{x}_i))$, which is same to the binary classification. If the discriminator is very

good, they will assign low scores to generated samples (negative samples) and assign high scores to true samples (positive samples). It will be hard to train a good generator making $p_G \approx p_{data}$, that could be a reason why GAN is hard to train. For discriminators, LSGAN changes the classification problem to regression problem (sigmoid to linear).

3.4.2 Wasserstein GAN (WGAN)

WGAN direct moves p_G to p_{data} by measuring the distance between p_G and p_{data}. Different from the objective function on f-GAN that is

$$\begin{aligned} V(G,D) &= \mathbb{E}_{x\sim p_{data}}[D(x)] - \mathbb{E}_{x\sim p_G}[f^*(D(x))] \\ V(G,D) &= \tfrac{1}{n}\textstyle\sum_{i=1}^{n} \log D(x_i) + \tfrac{1}{m}\sum_{i=1}^{m} \log(1-D(\tilde{x}_i)) \quad \text{orignal GAN} \end{aligned} \tag{11}$$

WGAN use the alternative objective function:

$$\begin{aligned} V(G,D) &= \mathbb{E}_{x\sim p_{data}}[D(x)] - \mathbb{E}_{x\sim p_G}[D(x)] \\ V(G,D) &= \tfrac{1}{n}\textstyle\sum_{i=1}^{n} D(x_i) - \tfrac{1}{m}\sum_{i=1}^{m} D(\tilde{x}_i)) \end{aligned} \tag{12}$$

where $D \in$ *1-Lipschize*. *K-lipschize* functions should satisfy $||f(x_1) - f(x_2)|| \leq K||x_1 - x_2||$, i.e., the change of inputs should less than the change of outputs if $K = 1$. If the discriminator function is *1-Lipschize*, it will be smooth and not too strong. The implementation of the constraints on $D(x)$ is weight clipping $to[-c,c]$.

However the weight clipping is not good, WGAN-GP is proposed using Gradient Penalty (GP) i.e., $||\nabla_x D(x)|| \leq 1$ for all x.

$$\begin{aligned} \max_D V(G,D) &\approx \max_D \mathbb{E}_{x\sim p_{data}}[D(x)] - \mathbb{E}_{x\sim p_G}[D(x)] - \lambda \textstyle\int_x \max(0, ||\nabla_x D(x)|| - 1)dx \\ &= \max_D \mathbb{E}_{x\sim p_{data}}[D(x)] - \mathbb{E}_{x\sim p_G}[D(x)] - \lambda \mathbb{E}_{x\sim p_{penalty}}[\max(0, ||\nabla_x D(x)|| - 1)] \end{aligned} \tag{13}$$

Here, $x \sim p_{penalty}$ can be sampled from the average p_{data} and p_G. The another variants is change max function to square function:

$$\max_D V(G,D) \approx \max_D \mathbb{E}_{x\sim p_{data}}[D(x)] - \mathbb{E}_{x\sim p_G}[D(x)] - \lambda \mathbb{E}_{x\sim p_{penalty}}[(||\nabla_x D(x)|| - 1)^2] \tag{14}$$

There are many algorithms how to do penalty sampling.

3.4.3 Energy-Based GAN (EBGAN)

It change the discriminator with auto-encoder, the input is the generated or true examples, the output is a scalar score. The score is high for true examples, and low for generated examples. Noted that energy margin is a trick.

3.5 Other Variants

3.5.1 InfoGAN

In original GAN, we will ranomly choose a vector, and given the vector, the generator generates examples, and the discriminator should judge if inputs is real or generated. However,

we cannot control the input to generate what we want. InfoGAN further divide the random input z of generator to (z, c), where z is the random noise and c is the set of features that are used to generate examples. So in addition, infoGAN add another decoder to predict the c given the generated examples x. The process is similar to autoencoder that makes $x \to z \to x$, but the difference is that infoGAN makes $(z, c) \to x \to c$.

3.5.2 VAE-GAN

VAE-GAN can be seen as VAE with an discriminator ($x \to z \to x$ + discriminator) or GAN with an encoder ($x \to z$ + GAN). In general, there are three parts, encoder ($x \to z$), decoder ($z \to x$), and discriminator ($x \to scalar$).

The objective of encoder is minimizing the reconstruction error and z is closer to normal distribution. The objective of decoder is minimizing the reconstruction error and fool discriminators. The objective of decoder is judge if the input is real, or generated examples.

Training is shown in Algorithm 3. For further improvement, the discriminator can do

Algorithm 3 the train on VAE-GAN

1: sample examples $x_1, ..., x_m$ from observed data
2: generate latent variable $\bar{z}_1, ..., \bar{z}_m$ for each x by encoder
3: generate examples $\bar{x}_1, ..., \bar{x}_m$ for each $\bar{z}$ by decoder
4: sample noise $z_1, ..., z_m$ from $p(z)$
5: generate examples $\hat{x}_1, ..., \hat{x}_m$ for each z by decoder
6: update encoder by decreasing $||\bar{x}_i - x_i||$ and decreasing $\mathrm{KL}(p(\bar{z}|x)||p(z))$
7: update decoder by decreasing $||\bar{x}_i - x_i||$ and increasing $D(\bar{x}_i)$ and $D(\hat{x}_i)$
8: update discriminator by increasing $D(x_i)$ and decreasing $D(\bar{x}_i)$ and $D(\hat{x}_i)$

3-classes classification on real, generated and reconstructed examples.

3.5.3 BiGAN

Different from VAE-GAN ($x \to z \to x \to scalar$), BiGAN separate encoder and decoder by making independent $x \to z$ and $z \to x$, i.e., the encoder output z is not the decoder input z. So the input of the discriminator is (x, z), and the objective is the same judging whether the (x, z) comes from encoder or decoder. We can put the input and the output of encoder as $p(x, \bar{z})$, and we can also put the input and output of decoder as $q(\bar{x}, z)$. The objective of encoder-decoder is to make p and q closer, and discriminator tries to discriminate them. The idea is similar to original GAN.

The training is shown in Algorithm 4. There is another variant called (tripleGAN). For the BiGAN, we can do some domain adversarial tasks.

Algorithm 4 the train on BiGAN

1: sample examples $x_1, ..., x_m$ from observed data
2: generate latent variable $\bar{z}_1, ..., \bar{z}_m$ for each x by encoder
3: sample noise $z_1, ..., z_m$ from $p(z)$
4: generate examples $\hat{x}_1, ..., \hat{x}_m$ for each z by decoder
5: update discriminator by increasing $D(x_i, \bar{z}_i)$ and decreasing $D(\hat{x}_i, z_i)$
6: update encoder and decoder by decreasing $D(x_i, \bar{z}_i)$ and increasing $D(\hat{x}_i, z_i)$

3.6 Applications

Take Photo Editing as an example. The task is that given an image and an constraints, we need to output the another image keep the most of information of the given image with the constraints (e.g., changing colors or shapes). The general idea is that we can encode the given image to hidden vectors z_0, move the hidden vectors slightly to another hidden vectors z in vector spaces with the constraints direction, and then generate the output image on z. The process is $x \to z_0 \to z \to \bar{x}$.

Auto-encoder can do $x \to z_0$ where the job of the encoders is to encode the x to hidden vectors z_0. $z_0 \to z \to \bar{x}$ is a generation problem, the standard GAN can be applied.

$$z^* = arg\ \min_z U(G(z)) + \lambda_1 ||z - z_0||^2 - \lambda_2 D(G(z)) \tag{15}$$

where (D, G) is the fully-trained GAN, and U is customized function to measure if the generated image satisfy the constraints. Other task like image resolution/completion using Conditional GAN.

3.7 Reinforcement Learning (RL) VS GAN in Generative Model

The main difference is that RL updates the generators from the human reward, and the GAN updates generators from the discriminator reward.

3.7.1 Maximum Likelihood vs RL

Suppose we want to build a model where the input is c and output is x. The maximum likelihood objective function is

$$CE_\theta = \frac{1}{N} \sum_{i=1}^{N} \log p_\theta(x_i|c_i) \tag{16}$$

and the gradients are

$$\nabla_\theta CE_\theta = \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \log p_\theta(x_i|c_i) \tag{17}$$

In RL, the objective function is to maximum the rewards:

$$\begin{aligned} R_\theta &= \textstyle\sum_c p(c) \sum_x R(x,c) p_\theta(x|c) \\ &= \mathbb{E}_{c \sim p(c), x \sim p_\theta(x|c)}[R(x,c)] \\ &\approx \tfrac{1}{N} \textstyle\sum_{i=1}^{N} R(x_i, c_i) \end{aligned} \tag{18}$$

where $R(x,c)$ is the reward from human metrics, and we can sample a lots of pairs of (x,c) to approximate the expectation. If we do sampling to approximate the expectation, the θ is reduced so that we cannot update θ directly by gradient. So step back (policy gradient):

$$\begin{aligned}\nabla_\theta R_\theta &= \sum_c p(c) \sum_x R(x,c) \nabla_\theta p_\theta(x|c) \\ &= \sum_c p(c) \sum_x R(x,c) p_\theta(x|c) \nabla_\theta \log p_\theta(x|c) \\ &= \mathbb{E}_{c\sim p(c), x\sim p_\theta(x|c)}[R(x,c)\nabla_\theta \log p_\theta(x|c)] \\ &= \frac{1}{N}\sum_{i=1}^{N} R(x_i,c_i)\nabla_\theta \log p_\theta(x_i|c_i)\end{aligned} \tag{19}$$

By comparing equations 17 and 21, we can see the RL add weight/reward, $R(x_i,c_i)$, to the gradient.

However, the $R(x_i,c_i)$ is the final reward. Sometimes, the reward is always positive. Although this, we can ignore the problem, because reward can be small positive and large positive. Anyway, we can add a baseline (threshold) to make the reward negative when they are below the baseline. So we can rewrite the equation 21 to be

$$\nabla_\theta R_\theta = \frac{1}{N}\sum_{i=1}^{N}[R(x_i,c_i) - b]\nabla_\theta \log p_\theta(x_i|c_i) \tag{20}$$

We generate thing by steps, we hope there is step-level rewards. Take sequential generation as example. The equation can be rewritten as

$$\begin{aligned}\nabla_\theta R_\theta &= \frac{1}{N}\sum_{i=1}^{N}[R(x_i,c_i) - b]\nabla_\theta \log p_\theta(x_i|c_i) \\ &= \frac{1}{N}\sum_{i=1}^{N}[Q(x_{ij},c_i) - b]\nabla_\theta \log p_\theta(x_{ij}|x_{i,<j},c_i)\end{aligned} \tag{21}$$

4 Reinforcement Learning (RL)

Given a trajectory $\tau = [s_1, a_1, s_2, a_2, ..., s_T, a_T]$, $p_\theta(\tau)$ is the probability of a trajectory τ given by a model parameterized by θ over all the possible trajectories. For a state s_t, the model will output an action a_t and get the reward r_t and $R(\tau) = \sum_{t=1}^{T} r_t$. So if the τ is given by the model, the expected reward of the model θ is:

$$\bar{R}_\theta = \sum_\tau p_\theta(\tau) R(\tau) \tag{22}$$

How to update the θ to maximize the reward $\bar{R}_\theta$. We can use the gradient ascent by $\theta \leftarrow \theta + \lambda \nabla_\theta \bar{R}_\theta$. The policy gradient is

$$\begin{aligned}\nabla_\theta \bar{R}_\theta &= \sum_\tau R(\tau)\nabla_\theta p_\theta(\tau) \\ &= \sum_\tau R(\tau) p_\theta(\tau)\nabla_\theta \log p_\theta(\tau) \\ &= \mathbb{E}_{\tau\sim p_\theta(\tau)}[R(\tau)\nabla_\theta \log p_\theta(\tau)] \\ &\approx \frac{1}{N}\sum_{n=1}^{N} R(\tau^n)\nabla_\theta \log p_\theta(\tau^n) \\ &= \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} R(\tau^n)\nabla_\theta \log p_\theta(a_t^n|s_t^n)\end{aligned} \tag{23}$$

There is an first-order assumption that action is outputted according to the current state. The general training process is that we can use the current model θ to generate many samples (s_t, a_t) and rewards r_t to update the model.

Problem 1 is that sometimes, the rewards are always positive, so we add a baseline to make the reward negative if it is below the baseline

$$\nabla_\theta \bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}[R(\tau^n)-b]\nabla_\theta \log p_\theta(a_t^n|s_t^n) \tag{24}$$

Problem 2 is that the reward $R(\tau)$ is the total reward of whole trajectory τ, and sometimes one action is bad but the total reward is positive or one action is good but the total reward is negative. So we change $R(\tau^n)=\sum_{t=1}^{T_n} r_t^n$ to be the reward from the current state to the end $\sum_{t'=t}^{T_n} r_{t'}^n$:

$$\nabla_\theta \bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}[\sum_{t'=t}^{T_n} r_{t'}^n-b]\nabla_\theta \log p_\theta(a_t^n|s_t^n) \tag{25}$$

Problem 3 is that the future rewards are not that important (empirically), so we add a decay to the future rewards, changing $\sum_{t'=t}^{T_n} r_{t'}^n$ to $\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n$ where $\gamma<1$:

$$\nabla_\theta \bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}[\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n-b]\nabla_\theta \log p_\theta(a_t^n|s_t^n) \tag{26}$$

usually, the baseline b is predicted by a neural network, and $R(\tau^n)$ is estimated by $A^\theta(s_t,a_t)$ that compute the future rewards using the model θ.

4.1 On-Policy to Off-Policy

All the things above is on-policy training. The general difference between on-policy and off-policy is that on-policy use the learned θ to interact with environment to obtain samples to train θ, while off-policy uses the another fixed θ' to obtain samples to train θ. In order to address the off-policy, we use the importance sampling strategy.

$$\mathbb{E}_{x\sim p(x)}[f(x)] = \mathbb{E}_{x\sim q(x)}[\frac{p(x)}{q(x)}f(x)] \tag{27}$$

Noted that the expectation is the same but the variances are different, i.e. $Var[f(x)] \neq Var[\frac{p(x)}{q(x)}f(x)]$. So the off-policy gradient is

$$\begin{aligned}\nabla_\theta \bar{R}_\theta &= \mathbb{E}_{\tau\sim p_\theta(\tau)}[R(\tau)\nabla_\theta \log p_\theta(\tau)] \\ &= \mathbb{E}_{\tau\sim p_{\theta'}(\tau)}[\frac{p_\theta(\tau)}{p_{\theta'}(\tau)}R(\tau)\nabla_\theta \log p_\theta(\tau)]\end{aligned} \tag{28}$$

If the action only depends on currents state, the gradient can be rewritten as:

$$\begin{aligned}\nabla_\theta \bar{R}_\theta &= \mathbb{E}_{(s_t,a_t)\sim\pi_\theta}[A^\theta(s_t,a_t)\nabla_\theta \log p_\theta(a_t|s_t)] \\ &= \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}[\frac{p_\theta(s_t,a_t)}{p_{\theta'}(s_t,a_t)}A^{\theta'}(s_t,a_t)\nabla_\theta \log p_\theta(a_t|s_t)] \\ &= \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}[\frac{p_\theta(a_t|s_t)p_\theta(s_t)}{p_{\theta'}(a_t|s_t)p_{\theta'}(s_t)}A^{\theta'}(s_t,a_t)\nabla_\theta \log p_\theta(a_t|s_t)] \\ &= \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t)\nabla_\theta \log p_\theta(a_t|s_t)] \quad p_\theta(s_t)\approx p_{\theta'}(s_t)\end{aligned} \tag{29}$$

So the objective function is total reward:

$$J(\theta)=\bar{R}_\theta = \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t)] \tag{30}$$

The off-policy is based on importance sampling, so we have to make θ and θ' close addressing variance issue. So the objective function should add regularization term to make the θ and θ' close:

$$J_{PPO}(\theta) = \bar{R}_\theta = \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t)] - \beta KL(\theta,\theta') \tag{31}$$

where $KL(\theta,\theta')$ makes the constraints on behaviors (actions a_t) not on parameters, which is called Proximal Policy Optimization (PPO) or Trust Region Policy Optimization (TRPO). The PPO/TRPO algorithm is shown in Algorithm 5. There is a improved PPO algorithm

Algorithm 5 PPO/TRPO algorithm

1: using θ' to interact with the environment to collect s_t, a_t and compute $A^{\theta'}(s_t,a_t)$
2: optimizing $J_{PPO}(\theta) = \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}[\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t)] - \beta KL(\theta,\theta')$
3: if $KL(\theta,\theta') > KL_{max}$, increase β
4: if $KL(\theta,\theta') < KL_{min}$, decrease β

called PPO2 by removing KL term with cliping strategy:

$$J_{PPO2}(\theta) = \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta'}}[\min(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A^{\theta'}(s_t,a_t), clip(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}, 1-\varepsilon, 1+\varepsilon)A^{\theta'}(s_t,a_t))] \tag{32}$$

where $clip(\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}, 1-\varepsilon, 1+\varepsilon)$ means if $\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} < 1-\varepsilon$, return $1-\varepsilon$ and if $\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)} > 1+\varepsilon$, return $1+\varepsilon$. Intuitively

- if $A^{\theta'}(s_t,a_t)$ positive and a_t is good, so we have to make $p_\theta(a_t|s_t)$ larger. If $p_\theta(a_t|s_t)$ is too small and $\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}$ is smaller than $1-\varepsilon$, we use $1-\varepsilon$ instead, but it cannot be larger than $1+\varepsilon$
- $A^{\theta'}(s_t,a_t)$ negative and a_t is bad, so we have to make $p_\theta(a_t|s_t)$ smaller. If If $p_\theta(a_t|s_t)$ is too larger and $\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}$ is larger than $1+\varepsilon$, we use $1+\varepsilon$ instead, but it cannot be smaller than $1-\varepsilon$

4.2 Q-learning

Different from on-policy and off-policy that aim to learn a policy, Q-learning is value-based method that aims to learn a critic to evaluate how good the current state is. Firstly, we define $V^\pi(s_t)$ as the critic to evaluate how good the actor π in the current s_t. The actor outputs a action given a specific state, i.e., $a = \pi(s)$. For example, V^π is a neural network where the input is state s and the output is a scalar (score) to say how many scores we will obtained after visiting s.

There are two method to estimate $V^\pi(s)$. 1) MC-based approaches. We compute the reward G_t until the end of interactions, $s_t \to V^\pi \to V^\pi(s_t) \to G_t$. 2) Temporal-difference (TD) approaches. We don't have to play to the end, and given a sample s_t, a_t, r_t, s_{t+1}, $V^\pi(s_t) - V^\pi(s_{t+1}) = r_t$. For example, we can use neural network to obtain $V^\pi(s_t)$ and $V^\pi(s_{t+1})$, and then, we can update the neural network V^π by regression to the difference r_t. MC methods have high variance while TD methods might be inaccurate.

Alternative to $V^{\pi}(s)$ that only consider the state, we define a function $Q^{\pi}(s,a)$ that is accumulated reward expects to be obtained after taking action a at state s. Given $Q^{\pi}(s,a)$, we can find a π' better than π. The better means $V^{\pi'}(s) \geq V^{\pi}(s)$ for all state. How to find π'? for each state we take the action $\pi'(s) = arg\ \max_a Q^{\pi}(s,a)$. The proof is that

$$\begin{aligned} V^{\pi}(s) = Q^{\pi}(s,\pi(s)) &\leq \max_a Q^{\pi}(s,a) \\ &= Q^{\pi'}(s,\pi'(s)) \\ &= E[r_t + V^{\pi}(s_{t+1})|s_t = s, a_t = \pi'(s_t)] \\ &\leq E[r_t + Q^{\pi'}(s,\pi'(s_{t+1}))|s_t = s, a_t = \pi'(s_t)] \\ &..... \\ &\leq V^{\pi'}(s) \end{aligned} \tag{33}$$

Usually, TD method is used to train Q function. But there are some problems.

Problem 1. If we use TD to train model, we have to do regression both on the $Q^{\pi}(s_t,a_t) = r_t + Q^{\pi}(s_{t+1},\pi(s_{t+1}))$ and $Q^{\pi}(s_{t+1},\pi(s_{t+1})) = Q^{\pi}(s_t,a_t) - r_t$. The both side of the equation are changed that makes the training hard and not converged. We fix one value as target and only update Q function by regression on one, i.e., $Q^{\pi}(s_t,a_t) = r_t + Q^{\pi}(s_{t+1},\pi(s_{t+1}))$ or $Q^{\pi}(s_{t+1},\pi(s_{t+1})) = Q^{\pi}(s_t,a_t) - r_t$ but not both. So at the beginning of the training, we will copy Q to $\hat{Q}$ that is fixed to produce target, and Q is updated. After several round updating, we will copy the updated Q to $\hat{Q}$ and continue training.

Problem 2. The Q-learning heavily depends on sampling. If states and actions are not sampled (not seen), we will always not make the unseen action. So if we do sampling, we have to take more explorations. 1) Epsilon Greedy. $a = arg\ \max_a Q(s,a)$ with a probability $1-\varepsilon$, otherwise a is random picked. 2) Boltzmann Exploration. $p(a|s) = \frac{exp(Q(s,a))}{\sum_a exp(Q(s,a))}$.

Problem 3. For one actor π, the samples are not much different. So we build a buffer to store the samples from different π, and replay (train) Q function by picking a batch from the buffer. This trick can make training samples diverse that similar to off-policy.

The training of Q function is shown in Algorithm 6 by considering the above 3 problems.

Algorithm 6 Q-learning algorithm

1: We initialize $Q = \hat{Q}$
2: Sample (s_t, a_t, r_t, s_{t+1}) based on Q (epsilon greedy trick)
3: Store (s_t, a_t, r_t, s_{t+1}) to buffer
4: Sample (s_i, a_i, r_i, s_{i+1}) from buffer (replay trick)
5: $y = r_i + \max_a \hat{Q}(s_{i+1},a)$
6: update Q to make $Q(s_i,a_i)$ close to y (target network trick)
7: every C steps reset $\hat{Q} = Q$

4.2.1 More magics

Double DQN Q function always over-estimate the rewards, because $Q(s_t, a_t) = r_t + \max_a Q(s_{t+1}, a)$ which tends to select action that is over-estimated. So Double DQN use

$$Q(s_t, a_t) = r_t + Q'(s_{t+1}, arg\ \max_a Q(s_{t+1}, a)) \tag{34}$$

If Q over-estimate action a, a is not over-estimated by Q', and also if Q' has an over-estimated action a, a might be not chosen by Q. In fact, in the target network trick, target network is regarded as Q'.

Dueling DQN Dueling DQN decompose $Q(s,a) = V(s) + A(s,a)$ with the constraint $\sum_a A(s,a) = 0$. If we want to update $Q(s,a)$ according to a sample s_t, a_t, we are more likely to update the value of $V(s)$, resulting all other actions in $Q(s,a)$ is updated, because rewards on all actions are added by $V(s)$.

Prioritized Replay The original replay take a buffer to train the Q function with used samples. However, these samples in buffer have different importances. Prioritized Replay regards samples having high TD errors should be used in training more times (high priority).

Multi-step In TD training, we sample (s_t, a_t, r_t, s_{t+1}), while in MC training, we sample from current state to the end. We are looking for a balance between TD training and MC training. We can look multiple steps when sampling, i.e., (s_t, a_t, r_t, ..., s_{t+N}), and $a_{t+N} = arg\ \max_a Q(s_{t+N}, a)$. We do temporal difference on multiple steps like $Q(s_t, a_t) = \sum_{t'=t}^{t+N} r_{t'} + Q(s_{t+N}, a_{t+N})$.

Noisy Net The noisy net is trick for action exploration. Different from the original epsilon greedy and others that sample actions for each times step, result sampled actions come from different policy, Noisy Net adds noisy (Gaussian or others) to the parameter of Q function, resulting sampled actions come from the sample policy (because Q function with noisy is not changed during episode).

- Epsilon Greedy: $a = arg\ \max_a\ Q(s,a)$ with a probability $1 - \varepsilon$
- Noisy Net: $\tilde{Q}(s,a) = Q(s,a) + noisy$, and then $a = arg\ \max_a\ \tilde{Q}(s,a)$

Distributional Q-function $Q(s,a)$ is the accumulated reward expects to be obtained after seeing state s and a. The different distribution could have the same expectation (mean). Distributional Q-function aims to build the distributions for each action a for state s. The original method use bars tricks.

Ensemble The rainbow system applies all these tricks together to obtain strong Q-functions.

4.2.2 Continuous actions

In the Algorithm 6, we have to obtain the $\max_a \hat{Q}(s_{i+1}, a)$ and $a_{i+1} = arg\ \max_a \hat{Q}(s_{i+1}, a)$. But if the action space is continuous, it is hard to solve. We can do sampling from continues space or using gradient ascent to find the optimal a. However, these methods are inefficient. So we can let Q network outputs a vector $\mu(s)$, a matrix $\Sigma(s)$ and a scalar $v(s)$. Given a continuous a:

$$Q(s,a) = -(a-\mu(s))^T \Sigma(s)(a-\mu(s)) + v(s) \tag{35}$$

and $a = \mu(s) = arg\ \max_a \hat{Q}(s,a)$

4.3 Actor-Critic

We have the policy gradient method and Q-learning method. We can combine the two method together. The gradient of the rewards in policy gradient is:

$$\nabla_\theta \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T_n} [\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n - b] \nabla_\theta \log p_\theta(a_t^n | s_t^n) \tag{36}$$

where $\sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n$ is the output of th Q function according to the definitions, i.e.,

$$Q^{\pi_\theta}(s_t, a_t) = \sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n \tag{37}$$

, and the baseline b aims to make the step reward can be positive and negative. So b should be the means over all possible values of $Q(s_t, a_t)$, i.e,

$$V^{\pi_\theta}(s_t) = b \tag{38}$$

So the gradient of the rewards in actor-critic is:

$$\nabla_\theta \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T_n} [Q^{\pi_\theta}(s_t^n, a_t^n) - V^{\pi_\theta}(s_t^n)] \nabla_\theta \log p_\theta(a_t^n | s_t^n). \tag{39}$$

Because $Q^{\pi_\theta}(s_t, a_t) = E[r_t + V(s_{t+1})]$, the above equation can be rewritten as:

$$\nabla_\theta \bar{R}_\theta \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T_n} [r_t^n + V^{\pi_\theta}(s_{t+1}^n) - V^{\pi_\theta}(s_t^n)] \nabla_\theta \log p_\theta(a_t^n | s_t^n). \tag{40}$$

The training is shown in Algorithm 7. The useful tricks are 1) V function network and π network can share the parts of parameters, 2) exploration on smooth π. Asynchronous A2C (A3C) can calculate gradients on multiple CPU/GPU.

Algorithm 7 Actor-Critic algorithm

1: We initialize V, π
2: Sample data using π to environments
3: Update V function
4: Update π based on V function

4.3.1 Pairwise Derivative Policy Gradient

When we want to learn Q function using TD method, we have to regress the value on the best action given by $Q(s,a)$, $y = r_i + \max_a \hat{Q}(s_{i+1}, a)$. The argmax problem $a^* = argmax\ Q(s,a)$ during gradient update, particular in continuous action space.

We can use GAN similar method to build a actor as generator to generate action, and make Q function as discriminator to ensure the action is the best action. The Q-learning is modified to the Algorithm 8.

Algorithm 8 Q-learning algorithm with actor-critic

1: We initialize $Q = \hat{Q}$, $\pi = \hat{\pi}$
2: Sample (s_t, a_t, r_t, s_{t+1}) based on π (epsilon greedy trick)
3: Store (s_t, a_t, r_t, s_{t+1}) to buffer
4: Sample (s_i, a_i, r_i, s_{i+1}) from buffer (replay trick)
5: $y = r_i + \cancel{\max_a \hat{Q}(s_{i+1}, a)}\ \hat{Q}(s_{i+1}, \hat{\pi}(s_{i+1}))$
6: update Q to make $Q(s_i, a_i)$ close to y (target network trick)
7: update π to maximize $Q(s_i, \pi(s_i))$
8: every C steps reset $\hat{Q} = Q$, $\hat{\pi} = \pi$

4.4 Sparse Reward

If we want to complete a task that is very far from the current state, the reward is sparse. For example, if we want to have a good job, we have to learn some skills and take the right actions. The problem is that sometimes learning one skill or take a action could be assigned with a small positive reward, leading a large negative reward in the final goal (get a good job). The question is how to get the right reward for these actions that are far from the final objectives.

4.4.1 Reward Shaping

One solution is reward shaping, we manually design some reward function assigning reward to all the possible action that we could take. For example, if we take “study” action, we will be not happy (reward is negative), but it can lead to high positive reward in the future. So we need to shape the reward of taking “study" action to be positive. There are many research works on reward shaping.

Curiosity We can add a curiosity rewards to each action, $R(\tau) = \sum_t^T r_t + r_t^i$ where r_t^i is curiosity reward in the t step. For example, $r_t^i = \text{diff}(\hat{s}_{t+1}, s_{t+1})$ where $\hat{s}_{t+1} = f(a_t, s_t)$, showing that we use the network f to predict the next state $\hat{s}_{t+1}$, and the curiosity reward is the difference between the predicted $\hat{s}_{t+1}$ and the s_{t+1} in our current policy π. If the difference is high that means next state is hard to predict (the future is hard to determine), the reward is

high that encourages curiosity. However, the problem is that if we make high rewards to the uncertainty, we will ignore the important features. So instead of the states, we can predict the features of the state, and $r_t^i = \text{diff}(\phi(\hat{s}_{t+1}), \phi(s_{t+1}))$ where ϕ could be a small network (linear or non-linear transformation) that takes states and output the features of the states. How to learn the ϕ function, we can use another network that takes $\phi(s_t)$ and $\phi(s_{t+1})$ and outputs $\hat{a}_t$, ensuring $\hat{a}_t$ and a_t is as close as possible. Because s_t takes action a_t to reach s_{t+1}, and if the ϕs_t and ϕs_{t+1} can be used to recover a_t, we can say ϕ function outputs the main features of the states.

Curriculum Learning The basic idea of curriculum learning is that we first train model on the simple cases and then the model will be trained on harder cases. The question is how to identify which cases are simple or hard. For example, we can use reverse curriculum generation. Starts from the goal state, we sample several states around the goal states and pick the top-k nearest states, and then we generate another set of several states around the top-k nearest states, and so on. In the end, we can get the orders of the learning cases from simplest to hardest.

Hierarchical RL We can divide the task to several small task and for each task, we apply RL methods.

4.5 No Rewards (Imitation Learning)

Imitation learning is to teach model behavor according to the examples, though we do not have rewards.

4.5.1 Behavior Cloning

We can collect a lots of (s,a) and train a model $\pi(s)$ that takes s and outputs correct a. The problems are that the observation is limited, and that we cannot have global optimization. Noted that in RL, the rewards are the expectation of the whole trajectory. One more problem is that the model will not learn the main features that leads the correct actions.

4.5.2 Inverse Reinforcement Learning

Rewards can be learnt from the correct (s,a), The idea is really simple, we can learn the reward function from the expert trajectory τ, and use the reward function to predict trajectory $\bar{\tau}$. The objective is the rewards of the expert trajectory is always larger then the rewards of the predicted trajectory, i.e., $\sum_{n=1}^{N} R(\tau_n) > \sum_{n=1}^{N} R(\bar{\tau}_n)$.

5 Deep Learning

5.1 Forward and Backward Pass

A Neural Network is regarded as a computational graph, where each node is a parameter node or a intermediate node. If the nodes are the same, they will be depicted independently but share the same parameters. In forward pass, we can compute the parts of gradient of current node. In backward pass, we can collect the gradient from the terminal node (loss node) and sum up the gradients from the nodes that are independent but share the same parameters.

5.2 Basic Units

- **Convolutional Neural Network (CNN)** takes the high dimension tensors and outputs the low dimension tensors by representing one neutron with its neighbours.
- **Recurrent Neural Network (RNN)** takes a arbitrary-length sequence of tokens and outputs a hidden representation for each token. The vanilla Long Short-Term Memory (LSTM) transforms the current input x_i and previous hidden representation x_{i-1} to an new input z_i, an input gates i_i, a forget gates f_i and a output gate o_i. The new memory is obtained by the previous memory with a forget gate plus the new input with an input gate: $c_i = c_{i-1} \odot f_i + z_i \odot i_i$. The new hidden representation is obtained by the new memory with a output gate: $h_i = o_i \odot c_i$. Compared to LSTM, the vanilla Gated Recurrent Unit (GRU) uses less gates by combining the input gate and the output gate into an update gate g_i but renames the forget gate as a reset gate r_i. The new input z_i is obtained by the current input and the previous hidden representation with the reset gate: $z_i = f(x_i, h_{i-1} \odot r_i)$. The new hidden representation h_i is obtained by the interpolation between the new input and previous hidden representation with the update gate: $h_i = g_i \odot h_{i-1} + (1 - g_i) \odot z_i$.
- **Spatial Transformer** adopts the matrix to transform the input tensor by indices. For example, the value in (1,2) is transformed to (2,3). The (2,3) is obtained by a linear function that could be a neural network. Sometimes the output is a float not an integer, so we use the interpolation to train the model.
- Highway Network simplified from GRU by removing the time step input x_i and reset gate r_i. Different from GRU that encodes time-step tokens, Highway network aims to encode the single input with more layers. Because highway networks have no input in each time step, the update gate g_i and the new input z_i for each layer are obtained by the hidden representation h_{i-1} in the previous layer. The new hidden representation $h_i = g_i \odot h_{i-1} + (1 - g_i) \odot z_i$, which is the same to GRU.

- **Residual Network** is the another version of highway network. Differently, it does not use the controlled gate and uses multiple layers to obtain the new input z_i. $h_i = h_{i-1} + z_i$ where $z_i = f(...f(h_{i-1})..)$.

- **Grid LSTM** is similar to the vanilla LSTM. The vanilla LSTM take (c_i, h_i) and outputs (c_{i+1}, h_{i+1}). The grid LSTM takes (c_i, h_i, a_j, b_j) and outputs $(c_i, h_i, a_{j+1}, b_{j+1})$. The simple modification is that we just concatenate c_i and a_j and concatenate h_i and b_j. The original LSTM take $([c_i; a_j], [h_i; b_j])$ and outputs $([c_{i+1}; a_{j+1}], [h_{i+1}; b_{j+1}])$. Furthermore, we can do 3-D or 4-D LSTM by the simple concatenation.

- **Recursive Neural Network** is more general to recurrent neural network. Encoding a sequence of tokens in RNN can be regarded as encoding a tree with left branching. Recursive Neural Network encode the tree in bottom-up manner, usually in binarized trees. The problem is how to merge two nodes into the single node. For example, Recursive Neural Tensor network take two nodes with self-transformation first and then do non-linear merging. Matrix-vector Recursive Network split the each node into two part: one vector and one matrix.

- **Tree LSTM**

- **Attention**. In conditional generation, bad attention leads to over-generation (solved by regularization?)

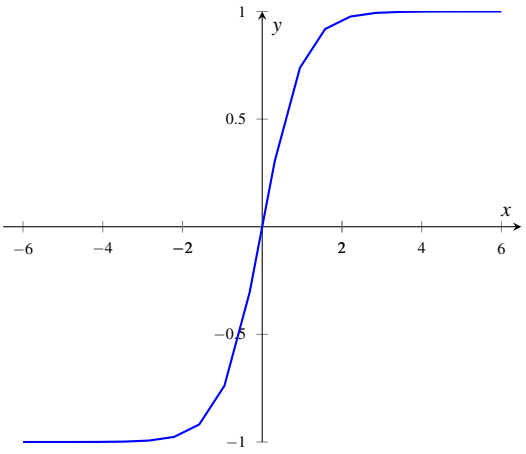
- **Memory Network** is originally used in document comprehension QA task. Documents are represented as vectors. Given a query vector, we extract relevant information from the document vectors to answer the questions. The extraction can be a loop, where extracted information can be concatenated to the query and then further extract information (hopping).

- **Neural Turing Machine** is similar to memory network, but it not only extracts the information from the memory, but also it can modify the memory. In details, for each query, they will output keys k, gates e and to-be-written a. They use k to compute the attention α for each part of memory and then use the attention to identify how many thing should be modified for each part. The memory will be modified: $m_i = m_i - \alpha_i(e \odot m + a)$

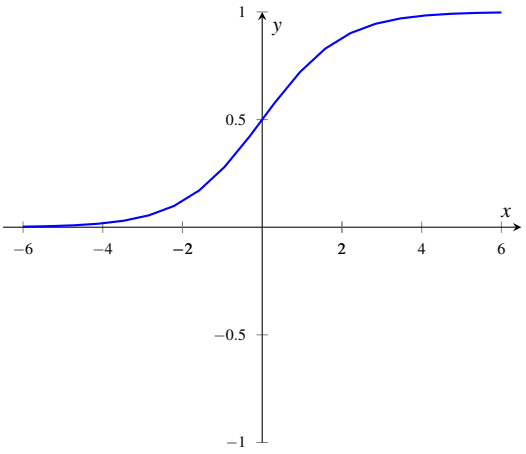
- **Pointer Network**

5.3 Activate Function

5.3.1 Tanh

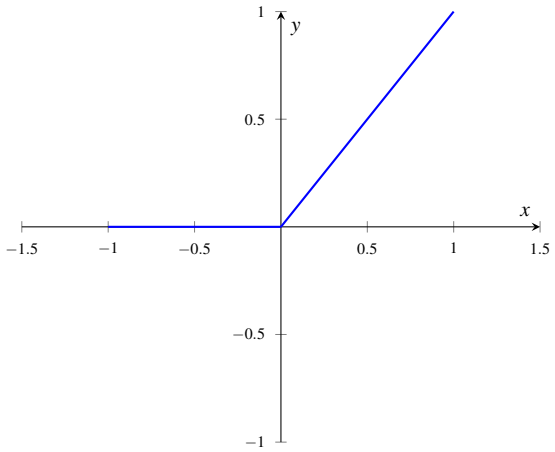


5.3.2 Sigmoid



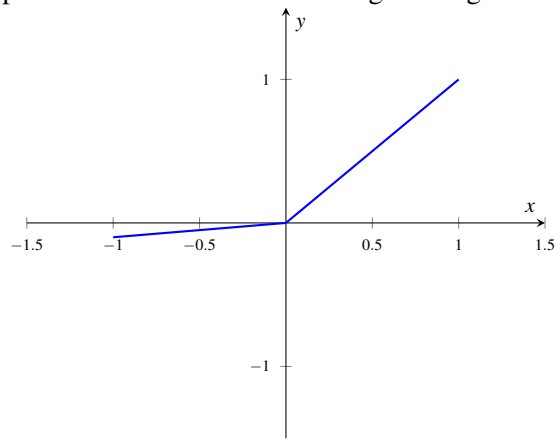
5.3.3 ReLU

1) fast to compute; 2) biology reason; 3) infinite sigmoid with different bias; 4) resolve gradient vanishing problem



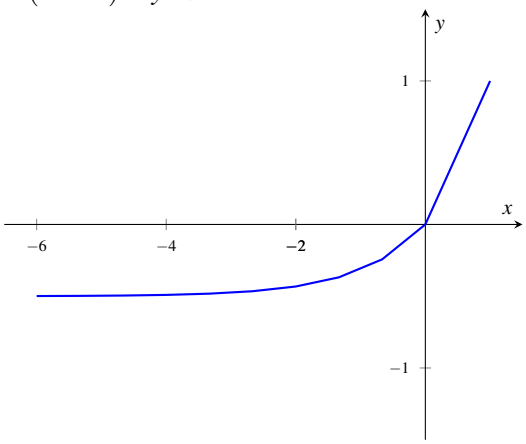
5.3.4 Parametric ReLU

$y = \alpha x$ if $y < 0$. It is reduced to leaky ReLU if $\alpha = 0.01$. It is Randomized ReLU if the α is sampled from the distribution during training.



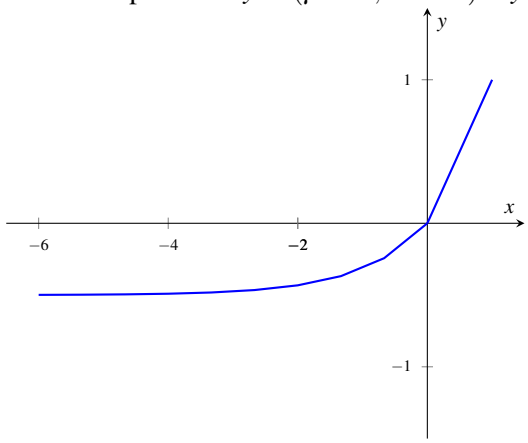
5.3.5 Exponential Linear Unit (ELU)

$y = \alpha(e^x - 1)$ if $y < 0$



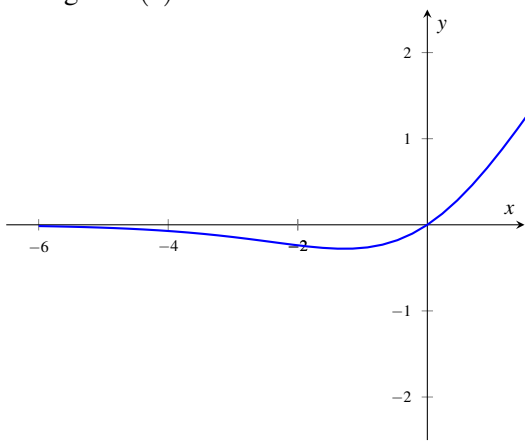
5.3.6 Scaled Exponential Linear Unit (SELU)

$y = \lambda\alpha(e^x - 1)$ if $y < 0$, $y = \lambda x$, where $\lambda = 1.0507009873...$ and $\alpha = 1.6732632....$ The numbers can be proved in the original paper. The idea is that the input value $x \sim (\mu = 0, \sigma^2 = 1)$, and the output value $y \sim (\mu = 0, \sigma^2 = 1)$ if $y = \text{SELU}(x)$.



5.3.7 Swish

$y = x * \text{sigmoid}(x)$



6 Machine Learning

MLE vs CE Maximum Likelihood equals to Minimize Cross Entropy [6]

The mismatch between training and test in language generation In training, we always have a reference, but in test, we generate the token conditioning on the predicted token in the previous step. RL and schedule modeling can help. RL uses loss for the whole sentence instead of the single token. Schedule modeling adopt sampling or beam search method to choose if we use the reference to train or the predicted token in the previous step to train.

[6] https://zhuanlan.zhihu.com/p/51099880

Batch Normalization (feature scaling) In deep learning, each dimension of a vector represent a feature by real number. Some dimensions have high value while some have low value. For the high-value dimension, the parameters are low-value, while the parameters are high-value if the value of the dimension is low. If we use a high learning rate, the training will be hard for the high-value parameters. A method is using a small learning rate, but it leads to slow training. Batch normalization is commonly used in batch-based training. The basic idea is to rescale the value of each dimension by means and variance. x_j^i is a scalar for the vector x^i, ith examples in a batch during training. $\mu_j = \frac{1}{N}\sum_i^N x_j^i$ and $\sigma_j = \sqrt{\frac{1}{N}\sum_i^N (x_j^i - \mu_j)^2}$.

$$\begin{aligned} \tilde{x}^i &= \text{BatchNormalization}(x^i, \mu, \sigma) \\ &= \frac{x^i - \mu}{\sigma} \end{aligned} \tag{41}$$

The benefit is avoiding covariant shift, and less exploding or vanishing gradients. For test, we don't have batch input. There are two options to get means and variance. 1) we can compute the means and variance for the whole training data with the final model. 2) we can collect all the means and variance during the entire training.

7 Meta-Learning

The meta-learning is method of learning how to learn in machine learning. Machine learning algorithms build a model based on sample data, known as "training data", in order to make predictions or decisions. But we have to manually design the model architectures, initialized parameters, evaluation metrics and so on. Meta-learning aims to automatically find these suitable settings, which should be manually set up in machine learning.

The basic idea is that given a set of (similar) tasks, we should automatically learn the machine learning settings that are good for all these tasks or other related tasks. General setting of the meta-learning is that, in training set, we have train data (support) and test data (query), in test set, we also have train data and test data. Meta-learning is task-level, and machine learning is model-level. In deep learning, meta-learning will have two sets of parameters, ϕ is a parameter of the neural network predicting the setting of the models with parameter θ.

Meta-learning is strongly related to few-shot learning, that each task has several training data. For example, 10-ways 5-shot classification is that we have 10 classes, and each class has 5 training examples.

7.1 Model-Agnostic Meta-Learning (MAML)

The purpose of MAML is to find the optimal initial parameters. MAML has a strong constraint that the two models have the same architecture.

The process of MAML: we use ϕ to initialize θ for each task, and in that task, we do the standard back-propagation training to get optimal $\hat{\theta}$, and collect all loss on the test data for

each task to be the loss on ϕ, $L(\phi) = \sum_{i=1}^{n} l^i(\hat{\theta}^i)$, $l^i(\hat{\theta}^i)$ is the model loss with $\hat{\theta}^n$ on the ith task. Noted that sometimes, $\hat{\theta}$ is obtained by ϕ with the one-step training. The motivation is that our purpose is to obtain the good initialized parameter ϕ that can be optimized in specific task within just one-step training.

How to get the gradient of ϕ from the tasks?

$$\begin{aligned} \nabla_\phi L(\phi) &= \nabla_\phi \sum_{i=1}^{n} l^i(\hat{\theta}^i) \\ &= \sum_{i=1}^{n} \nabla_\phi l^i(\hat{\theta}^i) \end{aligned} \tag{42}$$

Take one task as an example.

$$\nabla_\phi l(\hat{\theta}) = \begin{bmatrix} \partial l(\hat{\theta})/\partial \phi_1 \\ \partial l(\hat{\theta})/\partial \phi_2 \\ \vdots \\ \partial l(\hat{\theta})/\partial \phi_i \\ \vdots \end{bmatrix} \tag{43}$$

Take one item as an example.

$$\partial l(\hat{\theta})/\partial \phi_i = \sum_j \frac{\partial l(\hat{\theta})}{\partial \hat{\theta}_j} \frac{\partial \hat{\theta}_j}{\partial \phi_i} \tag{44}$$

where $\hat{\theta}_j = \phi_j - \varepsilon \frac{\partial l(\phi_j)}{\partial \phi_j}$, because we use the initialized parameter ϕ to do one-step training that obtain $\hat{\theta}$. If $i \neq j$,

$$\frac{\partial \hat{\theta}_j}{\partial \phi_i} = 0 - \varepsilon \frac{\partial l(\phi_j)}{\partial \phi_j \partial \phi_i} \approx 0 \tag{45}$$

else ($i = j$),

$$\frac{\partial \hat{\theta}_j}{\partial \phi_i} = 1 - \varepsilon \frac{\partial l(\phi_j)}{\partial \phi_j \partial \phi_i} \approx 1 \tag{46}$$

So 44 can be rewritten as

$$\partial l(\hat{\theta})/\partial \phi_i = \sum_j \frac{\partial l(\hat{\theta})}{\partial \hat{\theta}_j} \frac{\partial \hat{\theta}_j}{\partial \phi_i} \approx \frac{\partial l(\hat{\theta})}{\partial \hat{\theta}_i} \tag{47}$$

The gradient of ϕ is further step in test dataset. The real implementation is shown in Algorithm 9.

Algorithm 9 the train on MAML

1: sample a task i.
2: $\theta^i = \phi_{t-1} - \varepsilon \nabla_\phi l^i(\phi_{t-1})$
3: $\phi_t = \phi_{t-1} - \eta \nabla_{\theta^i} l^i(\theta^i)$

7.2 Reptile

Similar to MAML, Reptile is designed to find the optimal initialized parameters. Different from MAML, Reptile uses the average gradients of each training on the supprot data. Compared to MAML and pre-trained model. For example, on the ith task, we will get $\phi \rightarrow g_1 \rightarrow g_2$. The MAML use gradient of g_2 to update the ϕ, Reptile uses gradient of $g_1 + g_2$ to update the ϕ, and pre-trained model uses gradient of g_1 to update the ϕ.

7.3 Gradient by LSTM

Meta-learning can automatically learn the update of the parameters. Recall that the model parameters are updated by their gradient with a learning rate:

$$\theta_t = \theta_{t-1} - \eta \nabla_\theta l(\theta_{t-1}) \tag{48}$$

The memory update in LSTM is

$$c_t = f \odot c_{t-1} + i \odot z_t \tag{49}$$

where c_t is the memory in the tth step, f is the forget gate, i is the input gate, and z_t is the new input. If we make $c = theta$, $f = 1$, $z = -\nabla_\theta l(\theta)$, and $i = \eta$, the two equations are equal. We can use the meta-learning to learn the f and i in the LSTM architecture by back-propagation. In the real implementation, there is an assumption that parameters are independent, and the LSTM share to each model parameter.

Because we use the memory to represent the model parameter, i.e., $c = \theta$, the LSTM has no meory for the input. The variant model is to use another standard LSTM built below to capture the memory of the inputs. **One problem that I do not understand is that the loss comes from the test set, and is it reasonable?**.

Because the LSTM only focus on the one parameter (scalar), so we could use different architecture in test.

7.4 Metric-based Approach

In few-shot learning, we can directly feed all training data with test data to get the answers (Siamese Network). My thought: the drawback of few-shot learning is that the training data for each class is small (e.g., 1-shot), we could change the drawback to the pros. Because we do not have many training data, we can put all the training data as the part of the input. In machine learning, we use the large training data with label to learn a function f, and for new coming data, we can get the label by $f(test)$. In meta-learning, we directly model $f(train, test)$.

8 Life Long Learning

One model for all tasks. The challenge is that how to make the model perform well on the new task (transfer learning) without loss the performance on previous learned task (multi-task learning).

8.1 Elastic Weight Consolidation (EWC)

The learning is taken iteratively on each task. The new task loss function is

$$L'(\theta) = L(\theta) + \lambda \sum_i b_i(\theta_i - \theta_i^b)^2 \tag{50}$$

where the second term looks like the regularization

9 Linguistics-Compositionality

Proponents of compositionality typically emphasize the productivity and systematicity of our linguistic understanding. Compositionality is supposed to feature in the best explanation of these phenomena that if we understand some complex expressions we tend to understand others that can be obtained by recombining their constituents. Opponents of compositionality typically point to cases when meanings of larger expressions seem to depend on the intentions of the speaker, on the linguistic environment, or on the setting in which the utterance takes place without their parts displaying a similar dependence. They try to respond to the arguments from productivity and systematicity by insisting that the phenomena are limited, and by suggesting alternative explanations. (need more explanation) [7]

9.1 Definition

(C) The meaning of a complex expression is determined by its structure and the meanings of its constituents. For particular languages, **(C') For every complex expression e in L, the meaning of e in L is determined by the structure of e in L and the meanings of the constituents of e in L.** Compositionality entails (although on many elaborations is not entailed by) the claim that **syntax plus lexical semantics** determines the entire semantics for L.

Q1: whether there are such things as meaningful constituents? Q2: whether the meaningful constituents are independent? If the first question is positive, the answer to the second question is that constituents contribute different things to different thoughts, but these variable contributions, plus the way the constituents are combined, fully determine the meaning.

(C_{ref}) For every complex expression e in L, the reference of e in L is determined by the structure of e in L and the references of the constituents of e in L. The reference is not identical to the meaning (need more explanation). It is trivial that we can compositionally

[7] https://plato.stanford.edu/entries/compositionality/

assign something to each expression of a language (for example, if expressions serve as their own meanings, semantics is certainly compositional!) but it does not follow that it is trivial to adequately assign meanings to them. Zadrozny (1994) gives a method. Given a set S of strings generated from an arbitrary alphabet via concatenation and a meaning function m which assigns the members of an arbitrary set M (meaning) to the members of S, we can construct a new meaning function μ such that for all $s,t \in S$, $\mu(s.t) = \mu(s)(\mu(t))$ and $\mu(s)(s) = m(s)$. $m(s)$ returns the meaning of s. . is a binary operation (e.g., concatenation). What this shows is that we can turn an arbitrary meaning function into a compositional one, as long as we replace the old meanings with new ‘Ă‘IJmeanings‘Ă‘İ from which they are uniformly recoverable. What is not clear is whether these new “meanings" really are meanings

(C_{local}) For every complex expression e in L, the meaning of e in L is determined by the immediate structure of e in L and the meanings of the immediate constituents of e in L.

(C_{coll}) For every complex expression e in L, the meaning of e in L is determined by the structure of e in L and the meanings of the constituents of e in L collectively.

(C_{cross}) For every complex expression e in L, the meaning of e in L is functionally determined through a single function for all possible human languages by the structure of e in L and the meanings of the constituents of e in L.

(C_{stand})For every complex expression e in L, the standing meaning of e in L is determined by the structure of e in L and the standing meanings of the constituents of e in L.

(C_{occ}) For every complex expression e in L and every context c, the occasion meaning of e in L at c is determined by the structure of e in L and the occasion meanings of the constituents of e in L at c.

9.2 Other principles

- If two meaningful expressions differ only in that one is the result of substituting a synonym for a constituent within the other then the two expressions are synonyms.
- If two meaningful expressions differ only in that one is the result of substituting some synonyms for some constituents within the other then the two expressions are synonyms.
- To every syntactic rule corresponds a semantic rule that assigns meanings to the output of the syntactic rule on the basis of the meanings of its inputs.
- Complex expressions have their meanings in virtue of their structure and the meanings of their constituents.
- The meaning of an expression is determined by the meanings of all complex expressions in which it occurs as a constituent.

- The meaning of an expression is determined by the meaning of any complex expression in which it occurs as a constituent. This is a very strong thesis and most standard semantic theories are incompatible with it.
- The meaning of an expression is determined by the meanings of all expressions within any cofinal set of expressions. A cofinal set of expressions is a set such that any expression occurs as a constituent in at least one member of the set.
- L is compositional is often taken to mean that the meaning of an arbitrary complex expression in L is built up from the meanings of its constituents in L.

9.3 Formal Definition

Given a syntactic algebra is a partial algebra $\mathbf{E} = \langle E, (F_\gamma)_{\gamma\in\Gamma}\rangle$, where E is a set of expressions and every F_γ is a partial syntactic operation on E, and m is a meaning-assignment function, m is F-compositional if there is a function G on M:

$$m(F(e_1,...,e_k)) = G(m(e_1),...,m(e_k)) \tag{51}$$

So it can induce the semantic algebra $\mathbf{M} = \langle M, (G_\gamma)_{\gamma\in\Gamma}\rangle$, and it is a homomorphism between $\mathbf{E}$ and $\mathbf{M}$.

9.4 Arguments For Compositionality

Argument from productivity Since competent speakers can understand a complex expression e they never encountered before, it must be that they (perhaps tacitly) know something on the basis of which they can figure out, without any additional information, what e means. If this is so, something they already know must determine what e means. And this knowledge cannot plausibly be anything but knowledge of the structure of e and knowledge of the individual meanings of the simple constituents of e.

Argument from systematicity Anyone who understands a complex expression e and e' built up through the syntactic operation F from constituents $e_1,...,e_n$ and $e'_1,...,e'_n$, respectively, can also understand any other meaningful complex expression e'' built up through F from expressions among $e_1,...,e_n$, $e'_1,...,e'_n$. So, it must be that anyone who knows what e and e' mean is in the position to figure out, without any additional information, what e'' means. But the only plausible way this could be true is if the meaning of e determines F and meanings of $e_1,...,e_n$, the meaning of e' determines F and the meanings of $e'_1,...,e'_n$, and F and the meanings of $e_1,...,e_n$, $e'_1,...,e'_n$ determine the meaning of e''.

Although the arguments from productivity and systematicity are usually alluded to in the same breath, they are very different considerations. In systematicity, we have a function F. It makes a valid composition should satisfy the function F.

It seems to me that we have no such reason: semanticists have focused on whether they can hold on to compositionality while providing satisfactory explanations, not on whether they have to embrace compositionality in order to provide satisfactory explanations.

9.5 Arguments Against Compositionality

Putative counterexamples are always complex expressions whose meaning appears to depend not only on the meanings of their constituents and on their structure but on some *third factor* as well. I don't think these examples are conterexamples for compositionality. The main arugment is whether the lexical semantics (simple expressions) have some knowledges.