Preface

Machine learning (ML) is the study of algorithms that can learn from experience, to be brief. Recent years have witnessed a giant leap in ML, especially in *deep learning* based on neural networks. ML has some well-known applications, such as Siri, AlphaGo, ChatGPT, etc. The editor believes that ML, as the foundation of artificial intelligence, will be applied to more and more domains in the future, and engineers and researchers in all professions and trades should learn its basic ideas.

This book is a note of *Machine Learning 2021 spring*, a widely-honored course by Hung-yi Lee, Professor at National Taiwan University. Focusing on deep learning, this course can be one's first ML course, although it does not involve much classical, basic machine learning. Its webpage is https://speech.ee.ntu.edu.tw/~hylee/ml/2021-spring.html.

With 15 chapters, this book is expected to cover all important points taught in that course, even though that may make it a little missy. Many arXiv and other links are also included for reference. These chapters are not closely related, and the readers can just look for what they need.

The readers are expected to master modest amounts of basic calculus, linear algebra, probability, as well as programming (with Python). However, this book aims to give readers a rough but wide knowledge of ML, and mathematical rigor is considered secondary, nor will this book talk about how to program.

This book is written in Word. The notations follow the usual practice. Sentences and words in this book may lack authenticity attributed to the editor's poor English. Nor is this book carefully reviewed. For any possible inaccuracy or ambiguousness, please send an email to the author at lukeolivaw.ltc@gmail.com.

This book has been authorized by Prof. Lee. The videos of his course are available at https://www.youtube.com/c/HungyiLeeNTU, the YouTube channel of Prof. Lee.

Chapter 1 Introduction of Machine / Deep Learning

Machine Learning (ML) seems mysterious but, to put it simply, ML is to look for a function. For example, speech recognition is to find a function mapping audio to words, playing Go is to find a function mapping the current situation to the next move, and such functions are so complex that we can only find it with the aid of the machine.

There are different types of functions. In regression tasks, the function outputs a scalar. In classification tasks, the function is given options (i.e., classes) and outputs the correct one. (Go can be considered as a classification task with 19^2 classes.) Many think that a ML task is either regression or classification, but beyond these two kinds of tasks, there is *structured learning* which expects the machine to create something with structure (e.g., image or document).

There are three steps to do ML, i.e., to find a function.

Step 1, to define a function with unknown parameters. We need to decide the mathematical expression of a function with unknown parameters, based on our knowledge of the task (a.k.a. *domain knowledge*). Such a function is referred to as a *model*. The inputs of the function, which we have already known, are referred to as *features*. For example, $y = b + wx$ is a model, where x is a feature and w (a.k.a. *weight*) and b (a.k.a. *bias*) are unknown parameters to be learned from data.

Step 2, to define loss from training data. Loss is a function of the parameters (of the model), whose output represents how good a set of values for these parameters are. Still take the model $y = b + wx$ as an example. For the feature x_i, we get $y_i = b + wx_i$, and we know the true output value of the model is $\hat{y}_i$. ($\hat{y}$ is referred to as the *label*, and the training data consists of many feature-label pairs. Some people notate the model's output as $\hat{y}$ and the label as y, which is opposite to our notation and should be paid attention to.) Therefore, we can evaluate the difference between the model's prediction and the truth with $e_i = |y_i - \hat{y}_i|$, and define the loss $L = L(b, w) = \frac{1}{N}\sum_{n=1}^{N} e_n$. As we choose $e = |y - \hat{y}|$ to evaluate the error of the model, such an L is referred to as *mean absolute error* (MAE). We can also choose $e = (y - \hat{y})^2$, and the corresponding L is referred to as *mean square error* (MSE). The choice is based on the task, e.g., if y and $\hat{y}$ are probability distributions, we may choose *cross-entropy* (which we will see later).

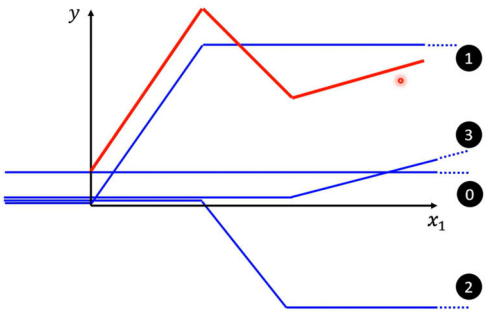
Step 3, to solve an optimization problem, e.g., $w^*, b^* = arg\min_{w,b} L$. That is to say, to find the set of values for the parameters which minimize the loss. The simplest way is *gradient descent*, and we will suppose that there is only one parameter w for convenience (The situation of multiple parameters is similar.). We pick an initial value w^0 (randomly, in common cases), compute $\frac{\partial L}{\partial w}\Big|_{w=w_0}$, and update w^0 to $w^1 = w^0 - \eta\frac{\partial L}{\partial w}\Big|_{w=w_0}$, where η is the *learning rate*. Intuitively, the learning rate decides how much the parameter changes at a time, and it is a *hyperparameter* which we decide (instead of learning from data). We update w in that way iteratively, until $\frac{\partial L}{\partial w}\Big|_{w=w_0} = 0$ （which rarely happens） or the number of iterations achieves the maximum we set (also a hyperparameter). Obviously, we will likely get a local minima of the loss, instead of the global minima, which we really want. However, that is not where the real problem lies when doing deep learning with gradient descent, although many think it is. (We will see the real problem later.)

These three steps, as a whole, is referred to as *training*. After training, our model achieves the smallest loss (ideally) on training data. However, to evaluate the model, we should compute the loss on data unseen during training.

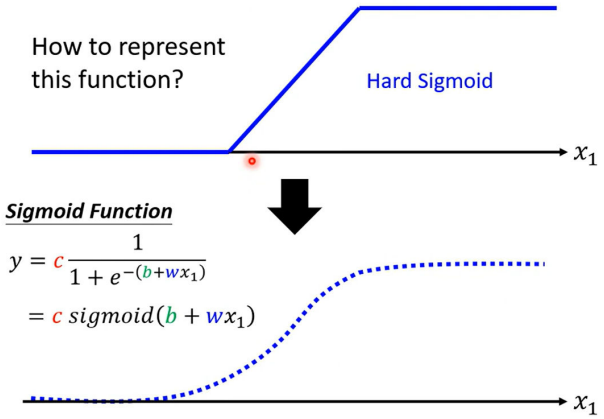
In one type of models, each dimension of the feature is multiplied with a weight and their sum is added with a bias, i.e., $y = b + \sum_j w_j x_j$. Such a model is referred to as a *linear model*. (The model we take as an example earlier, $y = b + wx$, is a linear model.) Obviously, linear models are too simple, and thus have severe limitations. Such limitation caused by the model is referred to as the *model bias* (It has no relation with the bias parameter b.) Therefore, we need a more

sophisticated, i.e., more flexible model.

We now define a function $y = f(x)$ whose value equals to 0 when x is less than a certain threshold, then increases, and stay the same when x is greater than another threshold. We can see that the sum of a constant and a set of such functions can represent all piecewise curves, as the example below shows. Intuitively, more pieces will require more such functions. Moreover, a continuous curve can be approximated by a piecewise linear curve, and we will have good approximation with sufficient pieces.



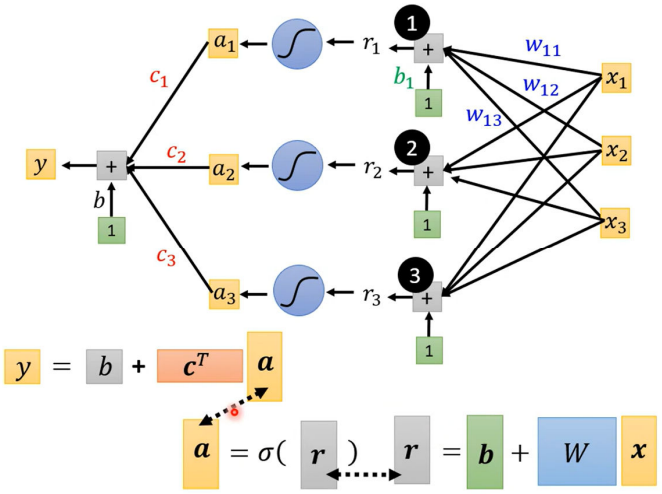
To represent the function $f(x)$, we may use the *sigmoid function* $\sigma(x) = \frac{1}{1+e^{-x}}$, and $f(x) = c\sigma(b + wx) = c\frac{1}{1+e^{-(b+wx)}}$. With different c, b and w, we get different curves. We can see that the sigmoid function is an approximation of $f(x)$, and $f(x)$ is also known as *hard sigmoid*, intuitively, as shown below.



Now we have a new type of models, $y = b + \sum_i c_i\sigma(b_i + w_i x)$ (compared with $y = b + wx$). For multiple features (To be precise, we should refer to it as a feature vector of certain dimension, instead of certain numbers of features. However, we will not make distinction.), we have $y = b + \sum_i c_i\sigma\left(b_i + \sum_j w_{ij}x_j\right)$ (compared with $y = b + \sum_j w_j x_j$). The whole process and its matrix representation are as demonstrated below. (The number of sigmoid functions is also a hyperparameter, and do not have to equal to the feature's dimension.)

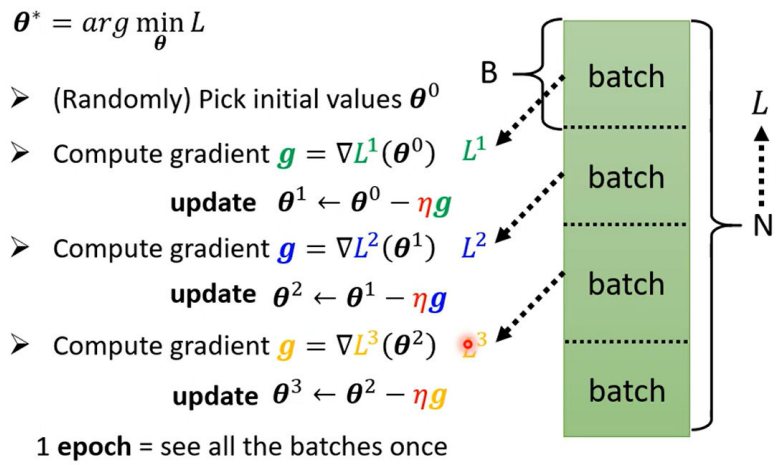
$$\begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} + \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\boldsymbol{r} = \boldsymbol{b} + W \quad \boldsymbol{x}$$

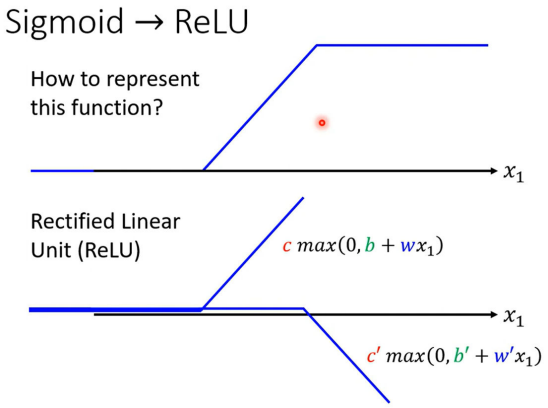


In a more general way, we can notate all unknown parameters as $\boldsymbol{\theta} = [\theta_1, \theta_2, \ldots]^T$. (Whether $\boldsymbol{\theta}$ is a row vector or a column one does not really matter.) Then, the loss of the new model can still be notated as $L = L(\boldsymbol{\theta})$, and the optimization problem is $\boldsymbol{\theta^*} = \boldsymbol{arg\,min_{\theta}\, L}$. Similarly, we update $\boldsymbol{\theta}^i$ to $\boldsymbol{\theta}^{i+1} = \boldsymbol{\theta}^i - \eta \boldsymbol{g}$, where $\boldsymbol{g}$ is the gradient $\nabla L(\boldsymbol{\theta}^i) = \left[\left.\frac{\partial L}{\partial \theta_1}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}^i}, \left.\frac{\partial L}{\partial \theta_2}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}^i}, \ldots \right]^T$.

In practice, we do not use all training data to compute the loss. Instead, we divide the data into several *batches* (a.k.a. *mini batches*), and use these batches one by one to compute the loss and update the parameters. When all batches are seen once, we call it an *epoch*, as shown below. (The reason why we use batches will be discussed later.) The *batch size* B is a hyperparameter. We should *shuffle* the training data after each epoch, i.e., the examples (feature-label pairs) in each batch are different in each epoch.

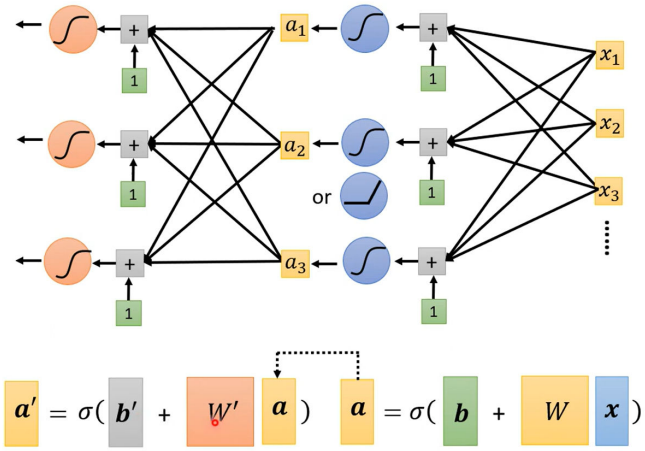


The model can vary. We may use hard sigmoid mentioned earlier, instead of sigmoid. The *Rectified Linear Unit* (ReLU) is defined as $g(x) = c\,max(0, b + wx)$, and hard sigmoid can be represented with the sum of two ReLU (with different c, b and w), as demonstrated below. Then, the model can be sketched as $y = b + \sum_{2i} c_i\, max\left(0, b_i + \sum_j w_{ij} x_j\right)$ (The subscript of the outer summation is $2i$, to show that we need two ReLU to replace a sigmoid.).



Both sigmoid and ReLU are referred to as the *activation function*, and there are other types of activation functions. ReLU is usually better than sigmoid, which will be explained in Chapter 2.

These models are more flexible than linear models, and can still be advanced. We may repeat what we have done for more times, i.e., to add more layers, as demonstrated below. (The number of layers is a hyperparameter.)



Now we can give our model a fancy name. The sigmoids or ReLUs are called the *neurons*, and the model is called the *neural network*, mimicking human brains seemingly. (The neurons shown above are connected to all neurons in the next layer, and such a network is referred to as *fully-connected neural network* (FNN).) In fact, dating back to 1980s and 1990s, the neural network is not a new technique, and does not have a good reputation. Researchers nowadays tend more to call these layers *hidden layers*, and the technique using many layers is also known as *deep learning*.

Models today often consist of hundreds of or even more layers. However, there is still one question: why do we choose to make the network "deep" instead of "fat", as we can approximate all continuous function with enough number of sigmoids or ReLUs? This question will be answered later.

The way to introduce deep learning in this chapter is not common. Another way of introduction is available at https://youtu.be/Dr-WRlEFefw.

Backpropagation is an efficient way to compute the gradient, c.f. https://youtu.be/ibJpTrp5mcE.

Chapter 2 Deep Learning

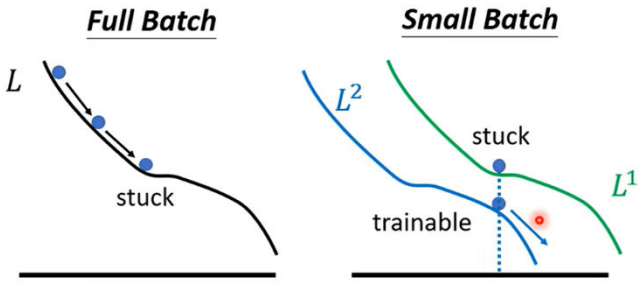
Small Batch vs Large Batch

We have seen optimization with batches in Chapter 1.

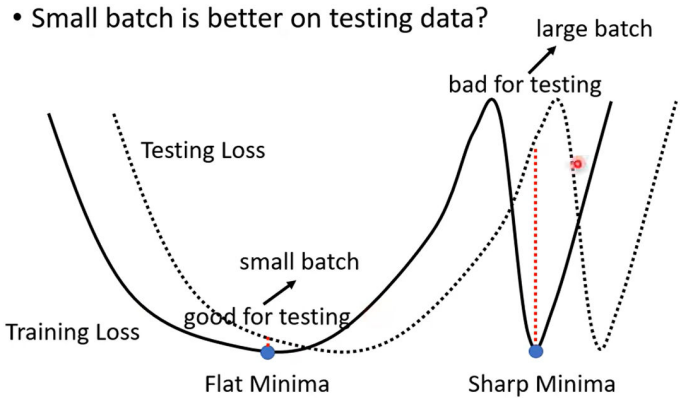
It seems that a larger batch size for optimization will take more time but be more powerful, while a smaller one will save time but be noisier (as it does not see all examples). However, with parallel computing, a larger batch size may not take longer time (although still limited by the competence of GPU), and a smaller batch size may require longer time for one epoch (as it needs more updates).

But that does not mean that a larger batch size is better. In fact, experiments show that a smaller batch size perform better, while training with a larger one may fail. A "noisy" update is better for training, as gradient descent is unlikely to be "stuck" when L differs, as demonstrated below

("Full Batch" here means considering all examples as a batch, i.e., no batch).



Moreover, even if we try to make the training accuracy of a model trained with a large batch size similar to that of one trained with a small batch size (by tuning hyperparameters, for example), the latter one still behaves better on testing data. As it is easier for a small batch to get out of a local minima, such training may probably end at a "flat" minima instead of a "sharp" one, which is better for testing, as shown below. (This is only one possible explanation.)



The difference between the small and large batch size is concluded as the table below. Obviously, the batch size is a hyperparameter.

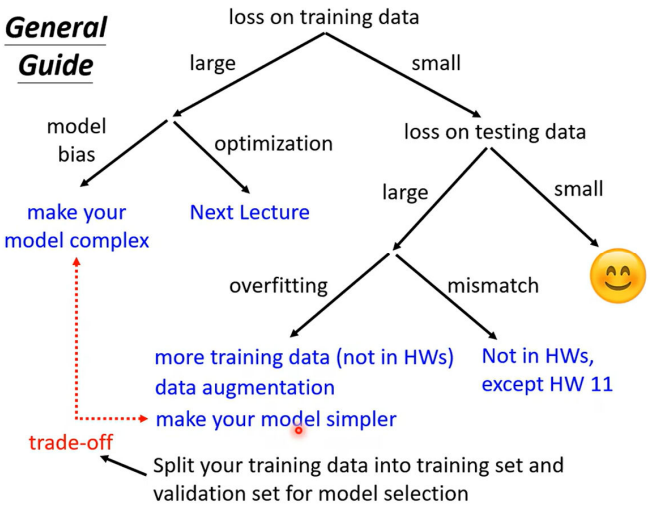
	Small	Large
Speed for one update (no parallel)	Faster	Slower
Speed for one update (with parallel)	Same	Same (not too large)
Time for one epoch	Slower	Faster
Gradient	Noisy	Stable
Optimization	Better	Worse
Generalization	Better	Worse

There may be methods to combine the advantage of small and large batch size, c.f. Large Batch Optimization for Deep Learning: Training BERT in 76 minutes, https://arxiv.org/abs/1904.00962; Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes, https://arxiv.org/abs/1711.04325; Stochastic Weight Averaging in Parallel: Large-Batch Training That Generalizes Well, https://arxiv.org/abs/2001.02312; Large Batch Training of Convolutional Networks, https://arxiv.org/abs/1708.03888; Accurate, large minibatch sgd: Training imagenet in 1 hour, https://arxiv.org/abs/1706.02677.

General Guidance

Let us review the framework of ML first. We have training data $\{\boldsymbol{x}^1, \hat{y}^1, \boldsymbol{x}^2, \hat{y}^2, \ldots, \boldsymbol{x}^N, \hat{y}^N\}$, $\boldsymbol{x}^i$ referred to as the feature and $\hat{y}^i$ referred to as the label. We first define a function, a.k.a. the model, $f_{\boldsymbol{\theta}}(x)$, with unknown parameters $\boldsymbol{\theta}$. Then, we define the loss $L(\boldsymbol{\theta})$, and solve the optimization problem $\boldsymbol{\theta}^* = \boldsymbol{arg}\,\boldsymbol{\min_{\theta}} \boldsymbol{L}$. With $y = f_{\boldsymbol{\theta}^*}(\boldsymbol{x})$, we can label the testing data $\{\boldsymbol{x}^{N+1}, \boldsymbol{x}^{N+2}, \ldots \boldsymbol{x}^{N+M}\}$.

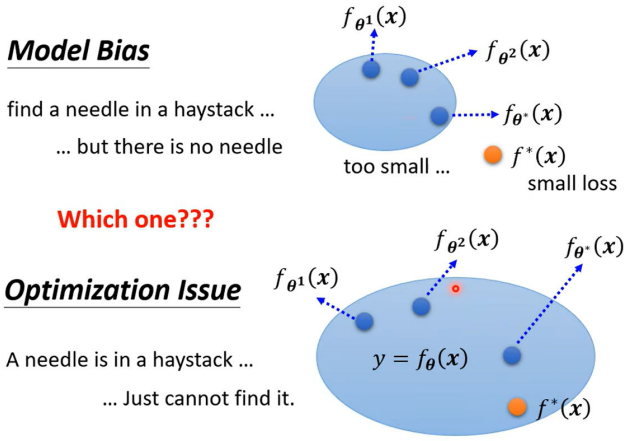
The general guidance to improve the model's performance is as shown below. (HW here refers to the homework of ML 2021, NTU.)



A large loss on training data may result from model bias. To be specific, the set of all possible functions $\{f_{\boldsymbol{\theta}^1}(\boldsymbol{x}), f_{\boldsymbol{\theta}^2}(\boldsymbol{x}), \ldots, f_{\boldsymbol{\theta}^N}(\boldsymbol{x})\}$ are so small that, even though we have found $f_{\boldsymbol{\theta}^*}(\boldsymbol{x})$, the loss is still too large. It is just like to look for a needle in a haystack but there is no needle. In this case, we should redesign the model to make it more flexible. We may use more features, or use more neurons and layers.

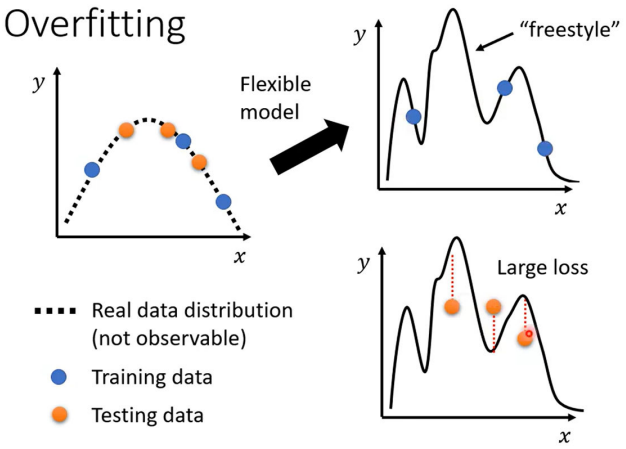
The large training loss does not always imply model bias. It may be caused by optimization issue, just like that there is a needle in a haystack but we cannot find it. The solution to it will be discussed later.

These two reasons of large training loss are as demonstrated below. We may distinguish them from comparison. We may start from shallow networks (or other models, e.g., linear models or *support vector machine*), which are easy to train. Then, if deeper networks do not obtain smaller loss on training data, there is an optimization issue, obviously.



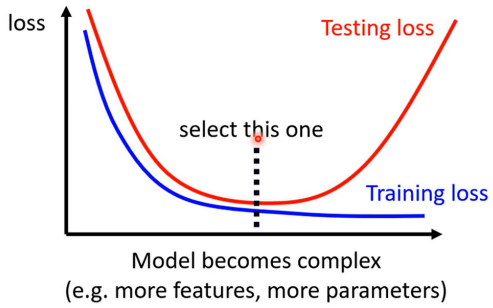
Having obtained small loss on training data, we should turn to the loss on testing data. If it is also small, our model is OK. If not, there may be other problems.

One reason is *overfitting*. That is to say, a flexible model may try to fit the training data with a "strange" curve, which lead to large loss on testing data. The phenomenon of overfitting is as shown below, and we will not talk about its mathematical explanation.



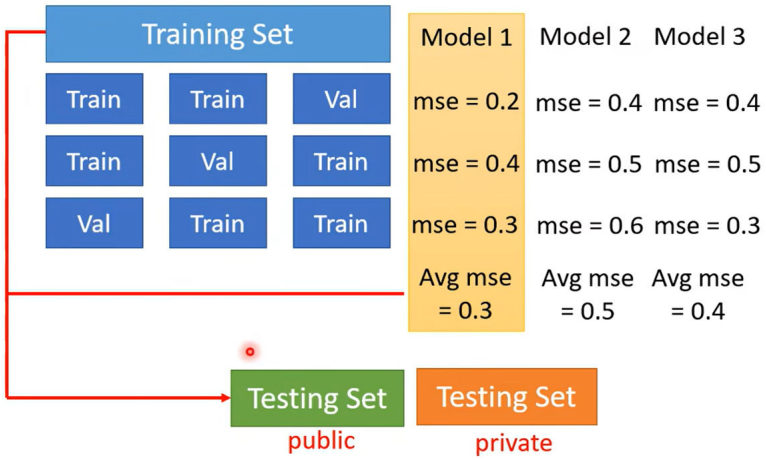
One easy but powerful solution is to use more training data. With no access to more training data, we can do data augmentation, i.e., "create" data with our domain knowledge. (For example, we may zoom an image or turn it left side right in image classification tasks.) Moreover, we may constrain the model by using fewer parameters, sharing parameters, using fewer features, early stopping, regularization, dropout, etc. (These methods will be introduced later.).

As the model becomes more complex (e.g., more features or parameters, not easy to define), the training loss will decrease, while the testing loss first decrease and then increase. Therefore, we want to find the model of the smallest testing loss, as shown below, which is referred to as bias-complexity tradeoff.



To select one, we need to evaluate the models. Obviously, it is meaningless to evaluate them on the training set, as good performance on those data does not imply good performance on the testing set or real data. (That may explain why machine usually beats human on benchmark corpora, which is public.) One possible method is *cross validation*. That is, we split the training data into the *training set* and the *validation set* (e.g., use 90% of all data as the training set and the rest validation set), train several models on the training set, and evaluate them with their performance (i.e., loss) on the validation set. We should select the model according to the performance on the validation set, instead of public testing data, in case of overfitting.

The training data is split randomly in the method above. However, *N-fold cross validation* is a better way. We split the training data into N folds of the same size, and train models for N times, using one different fold as the validation set and the rest as the training set. Then we select the model according to their average performance ($N-1$ on the training set and 1 on the validation set), as demonstrated below.



Another reason for small training loss and large testing loss is referred to as *mismatch*. It means that our training and testing data have different distributions, and it will not help to simply increasing the training data. (Some people think that mismatch is one kind of overfitting, but it is certainly especial.) We need some knowledge of the training and testing data to determine if mismatch happens.

When Gradient Is Small ...

We will talk about optimization and how to improve gradient descent in this section.

When the loss cannot get small enough (larger than that of a shallow model, or never decrease

from the beginning), the optimization probably fails.

Gradient descent may get "stuck" in the *critical points* where the gradient is 0. A critical point can be a local minima (or maxima), or a saddle point. As gradient will get "trapped" at a local minima while can still "escape" from a saddle point, it is meaningful to distinguish these two cases.
We do not know the whole graph of the loss. However, we can still approximate $L(\boldsymbol{\theta})$ around $\boldsymbol{\theta} = \boldsymbol{\theta}'$ with Taylor series approximation. To be specific, $L(\boldsymbol{\theta}) \approx L(\boldsymbol{\theta}') + (\boldsymbol{\theta} - \boldsymbol{\theta}')^T \boldsymbol{g} + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}')^T \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}')$, where $\boldsymbol{g} = \nabla L(\boldsymbol{\theta}')$ and $\boldsymbol{H}$ is the Hessian matrix, $H_{ij} = \frac{\partial^2}{\partial\theta_i \partial\theta_j} L(\boldsymbol{\theta}')$. At the critical point, $\boldsymbol{g} = \boldsymbol{0}$, and the term $\frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}')^T \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}')$ tells the properties of the critical point. For all vector $\boldsymbol{v}$ (i.e., for all vector $\boldsymbol{\theta} - \boldsymbol{\theta}'$), if $\boldsymbol{v}^T \boldsymbol{H} \boldsymbol{v} > 0$ (resp. $\boldsymbol{v}^T \boldsymbol{H} \boldsymbol{v} < 0$), $L(\boldsymbol{\theta}) > L(\boldsymbol{\theta}')$ (resp. $L(\boldsymbol{\theta}) < L(\boldsymbol{\theta}')$) around $\boldsymbol{\theta}'$, and the critical point is a local minima (resp. maxima). Otherwise ($\boldsymbol{v}^T \boldsymbol{H} \boldsymbol{v} > 0$ for some $\boldsymbol{v}$s and $\boldsymbol{v}^T \boldsymbol{H} \boldsymbol{v} < 0$ for some other $\boldsymbol{v}$s), the critical point is a saddle point. In other words, if $\boldsymbol{H}$ is positive definite (resp. negative definite), i.e., all its eigen values are positive (resp. negative), the critical point is a local minimum (resp. maxima); if some of $\boldsymbol{H}$'s eigen values are positive and some negative, the critical point is a saddle point.

Now we can deal with the saddle point. Suppose that $\boldsymbol{u}$ is an eigen vector of $\boldsymbol{H}$, then we know $\boldsymbol{u}^T \boldsymbol{H} \boldsymbol{u} = \boldsymbol{u}^T (\lambda \boldsymbol{u}) = \lambda \|\boldsymbol{u}\|^2$, where λ is the corresponding eigen value. Therefore, when $\boldsymbol{\theta}'$ is a critical point and the eigen value $\lambda < 0$, $L(\boldsymbol{\theta}' + \boldsymbol{u}) \approx L(\boldsymbol{\theta}') + \boldsymbol{u}^T \boldsymbol{g} + \frac{1}{2}\boldsymbol{u}^T \boldsymbol{H} \boldsymbol{u} = L(\boldsymbol{\theta}') + \frac{1}{2}\lambda \|\boldsymbol{u}\|^2 < L(\boldsymbol{\theta}')$. That is to say, we can update the parameters along the direction of $\boldsymbol{u}$ to escape the saddle point and decrease L.
However, the method above is seldom used in practice, as it needs too much computation. That being said, we have other methods to deal with saddle points, and the method above serves as a backup.

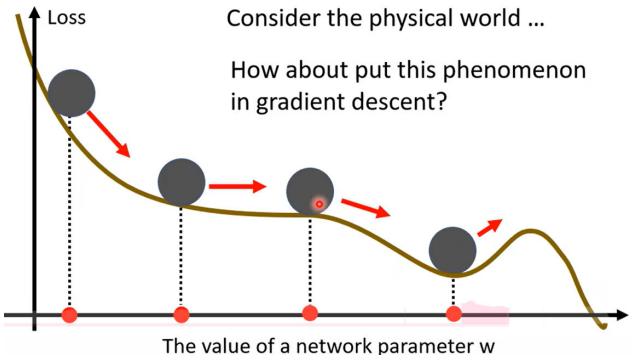
The local minima is difficult to deal with, as we can hardly get out of it. However, just like the story of Magician Diorena in *Death's End*, something sealed in 3-d space may not be sealed in higher dimensions. Similarly, we can assume that, as we have millions of parameters and the loss is thus in a high-dimension space, there are few local minimas. The result of experiments supports this assumption, and most critical points during training are saddle points, instead of local minimas.

Batch and Momentum

Momentum

Momentum is another method to deal with critical points.

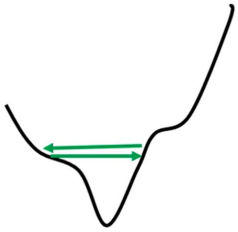
Imagining the error surface (i.e., the image of the loss function) as a surface in physical word and the parameters as a ball with inertia, we can find that such a ball is less likely to be stuck at a critical point, as shown below. That is the idea of momentum.



We have talked about (vanilla) gradient descent before. Gradient descent with momentum updates the parameters based on not only the (minus) gradient, but also the last movement. To be specific, we start at $\boldsymbol{\theta}^0$, and the initial movement is defined as $\boldsymbol{m}^0 = 0$; in each iteration, we compute the gradient $\boldsymbol{g}^i$, and movement $\boldsymbol{m}^{i+1} = \lambda \boldsymbol{m}^i - \eta \boldsymbol{g}^i$, so we update $\boldsymbol{\theta}^i$ to $\boldsymbol{\theta}^{i+1} = \boldsymbol{\theta}^i + \boldsymbol{m}^{i+1}$. (Pay attention that $\boldsymbol{\theta}$, $\boldsymbol{g}$ and $\boldsymbol{m}$ are all vectors.) In fact, the momentum (i.e., movement) is the weighted sum of all previous gradients.

Error Surface Is Rugged: Adaptive Learning Rate

The reason why the training get stuck is not necessarily that the parameters are around a critical point and the gradient is too small. (We may check the norm of the gradient.) However, the situation shown below can also lead to the same result. In fact, it is difficult for the parameters to get to around a critical point, and the critical point, although still need to be dealt with, is thus not the biggest problem in practice.



Training can still be difficult with no critical points, i.e., when the error surface is convex. On one hand, if we set too large a learning rate, the parameters may “keep jumping” between the “error canyon”, as shown above. On the other, if we set too small a learning rate, the training can hardly converge (i.e., to satisfy the condition of finishing training). Although there are other better methods to solve a convex optimization problem, we still need gradient descent to solve other, more complex optimization problems. Therefore, we must improve it.

We have seen that the learning rate cannot be one-size-fits-all, i.e., different parameters need different learning rates. We are going to focus on one single parameter for convenience, and it is easy to be generalized to more parameters. We will update θ_i^t (the subscript i means it is the

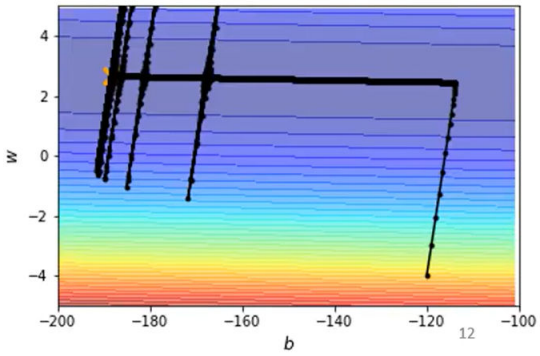
i th parameter) to $\theta_i^{t+1} = \theta_i^t - \frac{\eta}{\sigma_i^t} g_i^t$, where $g_i^t = \frac{\partial L}{\partial \theta_i}\Big|_{\boldsymbol{\theta}=\boldsymbol{\theta}^t}$, instead of $\theta_i^t - \eta g_i^t$. There are different methods to decide σ_i^t.

We may use the root mean square (RSE) of all the previous gradients (including the current one) as σ_i^t, i.e., $\sigma_i^t = \sqrt{\frac{1}{t+1}\sum_{j=0}^{t}\left(g_i^j\right)^2}$. This method is used in *Adagrad*. We can see that in this way, σ_i^t will be large (resp. small) when gradients are large (resp. small), and the learning rate will thus be small (resp. large)

We can see that the RSE method supposes that the gradients of one same parameter is similar, but that may not be true. The method *RMSProp* allows the learning rate to adapt dynamically. To be specific, σ_i^0 is defined as $\sqrt{\left(g_i^0\right)^2}$, and $\sigma_i^t = \sqrt{\alpha\left(\sigma_i^{t-1}\right)^2 + (1-\alpha)(g_i^t)^2}$ when $t \geq 1$, where α is a hyperparameter satisfying $0 < \alpha < 1$. Obviously, α indicates how much we care about the previous gradients.

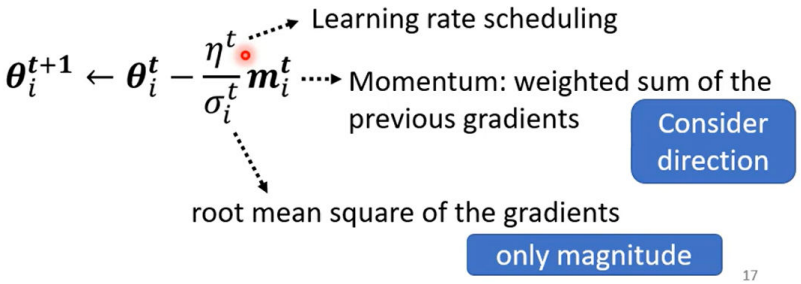
The most commonly used optimization method (a.k.a. *optimizer*) is *Adam*, which combines momentum and RMSProp (c.f. https://arxiv.org/pdf/1412/6980.pdf). (Adam is implemented in PyTorch, with a good set of hyperparameters.)

With dynamic learning rates, the optimization will less likely to get "stuck". However, there is a problem that, if many of the gradients of one certain parameter are small, the corresponding σ will get smaller and smaller, and thus lead to too large a learning rate. Luckily, that will not last forever because the gradient will probably become large when the parameters are far from critical points, as shown below.



Learning rate scheduling may help solve this problem. That is, we will update θ_i^t to $\theta_i^{t+1} = \theta_i^t - \frac{\eta^t}{\sigma_i^t} g_i^t$, instead of $\theta_i^{t+1} = \theta_i^t - \frac{\eta}{\sigma_i^t} g_i^t$, and schedule the learning rate by defining different η^ts. One method is called learning rate decay, which reduces the learning rate as training goes. Another method is called warm up (c.f. RAdam, https://arxiv.org/abs/1908.03265), where the learning rate increases first and then decrease. (Warm up has a long history dating back to 2015, but is used in many recent networks like transformer.)

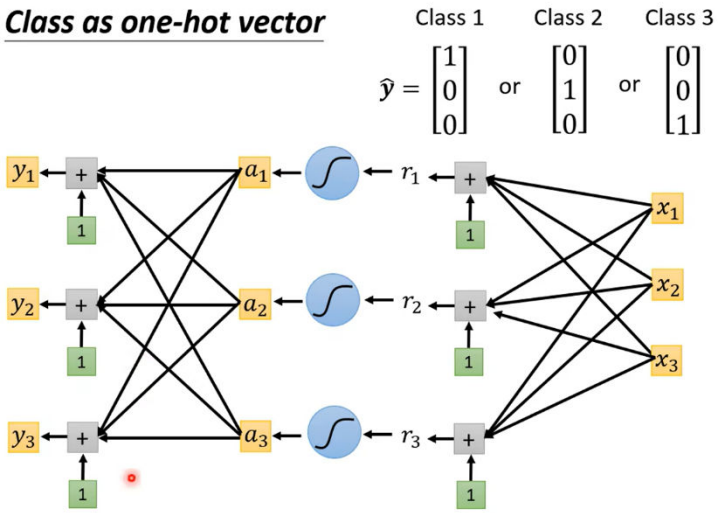
The methods of adaptive learning rate are as concluded below.



Loss also matters

We have talked about how to train a model for regression. Intuitively, we may consider classification as regression, i.e., encode class 1 with 1, class 2 with 2, etc., and use these numbers as labels. However, this method is not quite good, as it implies that class 2 is more similar than class 3 to class 1 (because 2 is closer than 3 to 1), which is wrong when the classes do not have such relationship.

Therefore, it is better to represent the class as a one-hot vector, i.e., a vector with an element 1 and the other elements all 0s. These vectors have a dimension (i.e., length) equals to the number of all the classes, and for class i, the ith element is 1. Naturally, the network should output a vector of the same length, i.e., scalars of the same number as the classes, as demonstrated below. The model is expected to output a vector as close to the label as possible.



Moreover, we will apply a function referred to as *Softmax* to the output vector (a.k.a. *logit*). The reason why we apply Softmax is complex, and we will not bother to talk about that. (A simple explanation is that Softmax normalizes outputs to values between 0 and 1, but it is not the whole truth.) Softmax takes y_is as input, and outputs $y_i' = \frac{exp\, y_i}{\sum_j exp\, y_j}$. Obviously, $0 < y_i' < 1$, $\sum_i y_i' = 1$, and moreover, the difference between y_is is enlarged. For binary classification, we can apply the sigmoid function instead, which is in fact equivalent.

To measure the difference e between $\boldsymbol{y}'$ and $\hat{\boldsymbol{y}}$, we can certainly use MSE, i.e., define $e = \sum_i \left(\hat{y}_i - y_i'\right)^2$. However, the *cross-entropy* is a more common way, defined as $e = \sum_i \hat{y}_i \, ln\, y_i'$. (It is

so commonly used that in PyTorch, Softmax is automatically built as the final layer of the network when calling cross-entropy.)

In fact, minimizing cross-entropy is equivalent to maximizing likelihood (although we are not going to talk about likelihood here), which is one of the reasons why we choose cross-entropy. The advantage of cross-entropy over MSE loss can be proved mathematically. To put it simple, the gradient of MSE loss is very small when the loss is large, causing difficulty for optimization, but that will not happen when using cross-entropy.

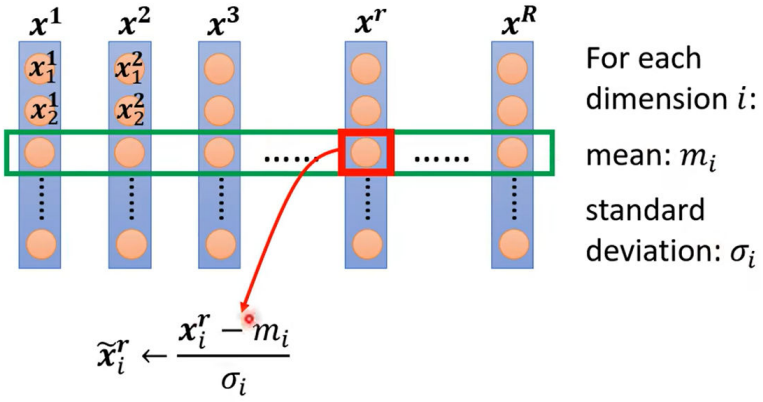
From this example, we can see that changing the loss function can change difficulty of optimization.

Batch Normalization

We have talked about how to help optimization using adaptive learning rate. Intuitively, to change the "landscape" of the error surface can also help.

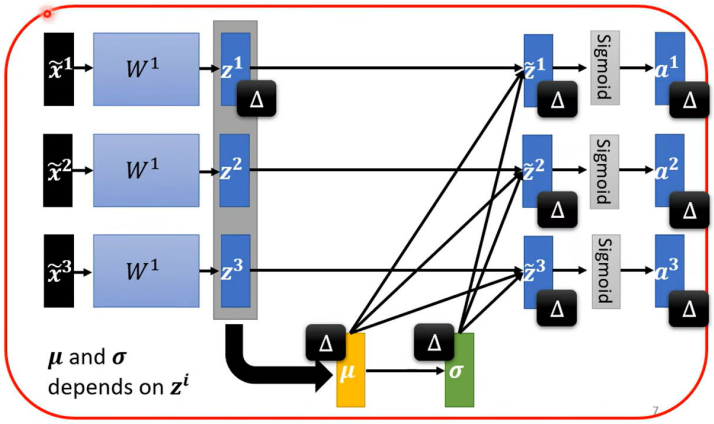
The training will be difficult if the scales of the slopes of different directions greatly vary. One possible reason of such a situation is that the scales of the feature's different dimensions greatly vary. Therefore, we want to "move" them into the same range, which can be done with *feature normalization*.

Notate the ith dimension of the jth feature as x_i^j. For each dimension i, we compute the mean m_i and standard deviation σ_i, and replace x_i^j with $\tilde{x}_i^j = \frac{x_i^j - m_i}{\sigma_i}$, as demonstrated below. We can see that the means of all dimension are all 0 and the variances all 1 now. This operation is referred to as *standardization*, one type of normalization.



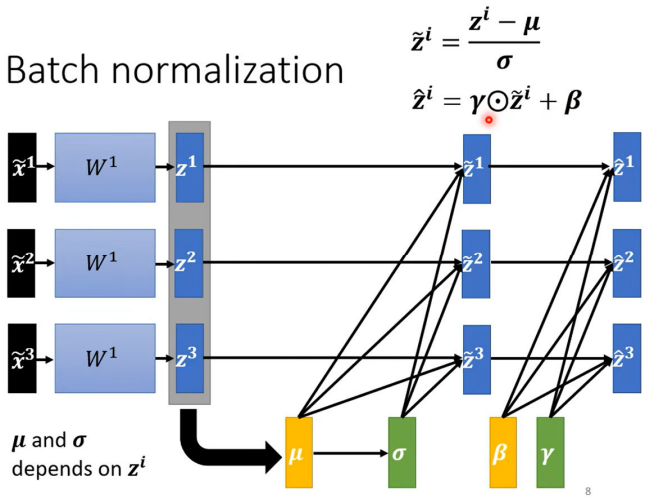
For deep learning, we need to normalize not only the input features, but also the inputs of hidden layers. It usually shows little difference in practice to normalize before or after the activation function (i.e., to normalize $\boldsymbol{z}^j$ or $\boldsymbol{a}^j$. For sigmoid, it may be better to normalize before the activation function.).

However, we can see that when one input of a hidden layer changes, all the outputs of this layer will change (as the mean and standard deviation change). Therefore, we have to consider a large network that takes many inputs and have many outputs, instead of one input and one output, as demonstrated below.



The memory of GPUs is limited, so we do not consider all examples as a whole. Instead, we consider a batch at a time, which is referred to as *batch normalization*. It is more suitable for a larger batch size, as such a batch can better approximate the distribution of the corpus (i.e., all examples).

We can see that the inputs of all hidden layers have a mean of 1 after applying normalization, which may put a limit on the network. Therefore, we often add another step after normalization. We compute $\hat{\boldsymbol{z}}^j = \boldsymbol{\gamma} \odot \tilde{\boldsymbol{z}}^j + \boldsymbol{\beta}$, where $\boldsymbol{\gamma}$, $\boldsymbol{\beta}$ are another two parameters to learn, as demonstrated below. That will not cause the scales of different dimensions to vary greatly, as we initialize $\boldsymbol{\gamma}$ as a one-vector (elements are all 1) and $\boldsymbol{\beta}$ as a zero-vector.

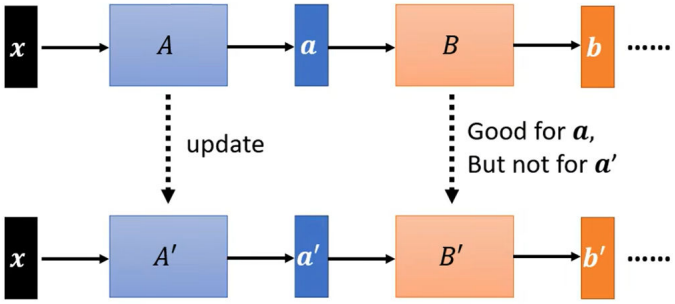


However, we do not always have a batch of data at testing (a.k.a. inference) stage (We cannot wait until we have inputs of the batch size to give an output!). We compute the moving average of the mean $\boldsymbol{\mu}$ and the standard deviation $\boldsymbol{\sigma}$ of the batches during training. To be specific, we get $\boldsymbol{\mu}^t$ after training on the tth batch, and update $\overline{\boldsymbol{\mu}}$ to $p\overline{\boldsymbol{\mu}} + (1-p)\boldsymbol{\mu}^t$ (The first $\overline{\boldsymbol{\mu}}$ is defined as $\boldsymbol{\mu}^1$.), where p is a hyperparameter ($p = 0.1$ in PyTorch).

For more about batch normalization, c.f. https://arxiv.org/abs/1502.03167.

Some researchers think batch normalization helps to deal with “internal covariate shift” (All the layers are updated at the same time according to previous inputs, but a change in parameters will

change the inputs of following layers, so the new parameters for those layers may not be so good, as demonstrated below.), as it keeps the outputs similar to the previous ones (e.g., to keep $\boldsymbol{a}'$ similar to $\boldsymbol{a}$). However, experiments show that internal covariate shift hardly happens, nor would it affect training, c.f. How Does Batch Normalization Help Optimization, https://arxiv.org/abs/1805.11604.



Instead, experimental results (as well as theoretical analysis) support that batch normalization can change the landscape of the error surface, and thus helps training. In fact, there are many other methods that can achieve the same purpose (even better), and "the positive impact of batch normalization might be somewhat serendipitous" (said the author of How Does Batch Normalization Help Optimization).

There are other well-known methods for normalization, e.g., batch renormalization (c.f. https://arxiv.org/abs/1702.03275), layer normalization (c.f. https://arxiv.org/abs/1607.06450), instance normalization (c.f. https://arxiv.org/abs/1607.08022), group normalization (c.f. https://arxiv.org/abs/1803.08494), weight normalization (c.f. https://arxiv.org/abs/1602.07868), spectrum normalization (c.f. https://arxiv.org/abs/1705.10941).

Chapter 3 Convolutional Neural Network

From this chapter on, we will consider how to design network architecture. We will first talk about *Convolutional Neural Network* (CNN), a network architecture designed for image, and take it as an example, we will see how network architecture affects the network's performance.

Image classification is a certain kind of questions require the machine to figure out which kind of objects is in the images. We suppose that all the images to be classified have the same size in the following discussion. (If not, we can zoom them to the same size.)
We use a *one-hot vector* $\hat{y}$ to represent the ground-truth of classification. Therefore, the length, i.e., the dimension, of the vector equals the number of classes. The model outputs a vector y' of the same dimension. Loss function is the cross-entropy between $\hat{y}$ and y'.
The image input is in fact a 3-D tensor, whose 3 dimensions are the length, the height and the channels (3 channels R, G, B for a colorful image or 1 channel for a black and white one). One way to deal with it is to simply reshape it as a vector and use a fully-connected network, but such a network is unnecessary and can even lead to overfitting.

The question of image classifying has some characteristics according to which we can simplify the network.

Firstly, the machine should identify some critical patterns. (Perhaps humans also resort to the similar way.) A neuron does not have to see the whole image as some patterns are much smaller than the whole image.

Therefore, we can set a *receptive field*, which is a part of the image, for each neuron in the first layer, and these neurons only take its own receptive field as input. A receptive field is square and cover all 3 channels in most but not all cases. Different neurons' receptive fields can overlap or even be the same, and they can have different sizes and cover only some of the channels.

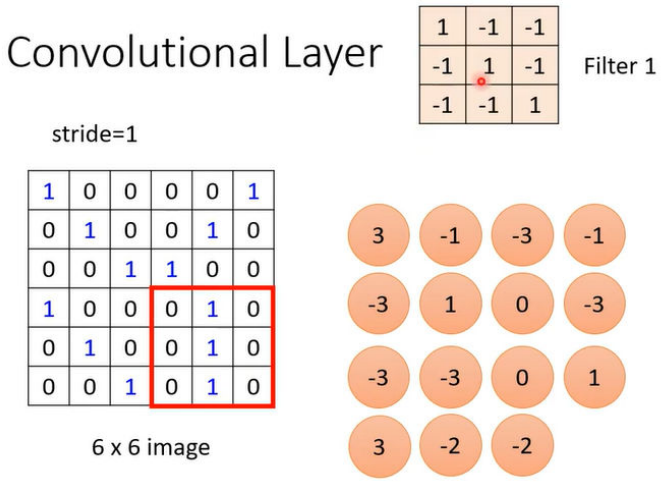
The typical setting of receptive fields is to include all channels and choose a length and a height as kernel size (e.g., 3×3), and use a set of neurons (e.g., 64 neurons) for each receptive field. The distance between two adjacent receptive fields is referred to as stride, which is a hyperparameter usually smaller than kernel size (e.g., 1 or 2). If a field exceeds the image, we should do padding by fill the "blank pixels" with 0 or the average value of all pixels. Thus, the receptive fields cover the whole image.

Secondly, it is obvious that the same patterns may appear in different regions of the image. It is unnecessary that each receptive field has its own neuron(s) to detect one same pattern.

Therefore, we need *parameter sharing*, which means several neurons have exactly the same $\boldsymbol{w}$ and $\boldsymbol{b}$. The typical setting is to use one same set of neurons for each receptive field. The parameters of each neuron of this set form a tensor, referred to as a filter. (We will look into filters from a different viewpoint later in this chapter.)

Now we have introduced two methods – receptive field and parameter sharing – and by using them together, we get *convolutional layer*. A network with convolutional layer is referred to as CNN. Compared with fully-connected network, intuitively, CNN has a larger bias and a smaller flexibility. CNN's larger model bias does not matter in regard to image. (But CNN may not work well in other situations.)

A more common understanding of CNN is as follows. There are a set of *filters* in a convolutional layer, each of them a $n \times n \times \mathrm{channel}$ tensor and detects one pattern in the image. The values in these tensors are unknown parameters. A filter is "placed" on the top left corner of the image first, and "convolves over" the image row by row, moving a distance referred to as stride each step. We calculate the Hadamard product of the filter and the part of image tensor "below" it, add a bias to the product, and get a matrix composed of these results finally. The process is as demonstrated below, noticing that the bias is not shown.



Repeating the process with all filters, we will get several matrices (the same number as filters). We call them feature map, which can be considered as a new image with channels the same number as filters. This "image" then serves as the input of the second layer.

We can see that the same size of filter in higher layers will receive a larger part of the original image, ensuring big patterns to be detected.

This understanding is no different from what we discussed earlier essentially. Moreover, we can also get the same result by adjusting the parameters of a fully-connected layer: the output of the layer $h_{i,j} = u_{i,j} + \sum_{k,l} w_{i,j,k,l} x_{k,l} = u_{i,j} + \sum_{a,b} v_{i,j,a,b} x_{i+a,j+b}$, where $v_{i,j,a,b} = w_{i,j,i+a,j+b}$; according to the two characteristics of image classifying mentioned above, $v_{i,j,a,b}$, $u_{i,j}$ should be irrelevant to i, j and the range of a, b should not be too big; therefore, we get $h_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} v_{a,b} x_{i+a,j+b}$. (Some people believe that, theoretically, such an operation is *cross-correlation*, instead of convolution.

There is one more way to simplify the network. Roughly speaking, subsampling the pixels does not change an image. We can do *pooling* on feature maps, which means we choose a square of a certain size (e.g., 2×2) and replace each such square (no overlap) in the feature map with one value. This value may be the maximum of the values in the square (Max Pooling), the minimum, the average, etc. We usually apply several convolutional layers, then a pooling layer, and repeat that.

Pooling can reduce the scale of calculation but has a negative effect on the network's performance. For this reason, quite a lot networks use fully convolutional network (no pooling layer) as computing power advances nowadays.

Pooling has no parameters, so we can consider it as a kind of activation function.

After several repeats of convolutional layer(s) and (maybe) a pooling layer, the tensor is flattened, i.e., reshaped to a vector, and goes through fully-connected layers and an activation function (e.g., Softmax). This is how a whole, typical CNN works.

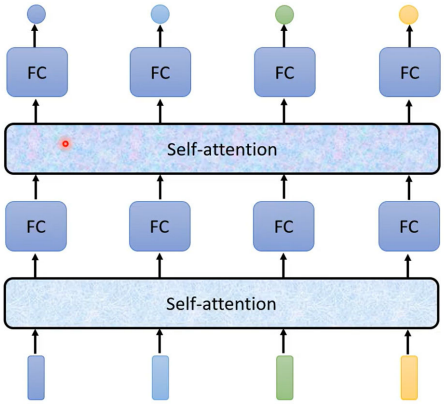
CNN is mostly used for image classification, but it has some other applications like Go Playing, which has similar characteristics to image. Obviously, the design of CNN should be adjusted for different tasks.

CNN is not invariant to scaling and rotation, so we need data augmentation. Spatial Transformer Layer can avoid this problem, but we will not discuss it here.

Chapter 4 Self-Attention

When input is not one vector but a set of vectors, of which the number may change, and that output is not one scalar but scalars, we have another network architecture named *self-attention*.
A common case of vector set as input is text processing. To represent words by vectors, we can apply one-hot encoding or *word embedding*, the latter one including semantic information, c.f. https://youtu.be/X7PH3NuYW0Q. Another case is audio processing. The audio information in a short period (25 ms usually), i.e., a frame, is represented by a vector, and two adjacent frames are at an interval of 10 ms usually. A graph can also be considered as a set of vectors (each node a vector).
In some cases, each vector has a label, e.g., *part-of-speech (POS) tagging*. In another some cases, the whole sequence has a label, e.g., *sentiment analysis*. In other cases, model decides the number of labels, also known as *sequence to sequence* (sec2sec), e.g., translation.
We will only talk about sequence labeling (each vector a label) in this chapter.

Fully-connected network can consider the context with a window including the vector and its neighbors, but can hardly deal with the whole sequence as its length varies and too big a window may lead to overfitting. That's why we introduce self-attention.
Self-attention can be described as follows: Self-attention layer outputs a vector for each input vector considering the whole input sequence. Then fully-connected layer outputs a label for each vector that self-attention layer outputs, which includes the context information. These two steps can repeat, as demonstrated below.



Suppose that the input of a self-attention layer (the origin input or another layer's output) is $\boldsymbol{a}^1, \boldsymbol{a}^2, \boldsymbol{a}^3, \ldots, \boldsymbol{a}^n$ and the corresponding output $\boldsymbol{b}^1, \boldsymbol{b}^2, \boldsymbol{b}^3, \ldots, \boldsymbol{b}^n$.To get $\boldsymbol{b}^1$, we need to find the vectors relevant to $\boldsymbol{a}^1$ in the sequence. We calculate a scalar α to measure the relevance between two vectors $\boldsymbol{a}^i, \boldsymbol{a}^j$, and α is mostly calculated as follows:

$$\boldsymbol{q}^i = \boldsymbol{W}^q \boldsymbol{a}^i$$
$$\boldsymbol{k}^j = \boldsymbol{W}^k \boldsymbol{a}^j$$

$$\alpha_{i,j} = \boldsymbol{q}^i \cdot \boldsymbol{k}^j$$

where $\boldsymbol{W}^q, \boldsymbol{W}^k$ are matrices. There are some other methods to get α, e.g.,

$$\alpha_{i,j} = \boldsymbol{W}\tanh\left(\boldsymbol{q}^i + \boldsymbol{k}^j\right)$$

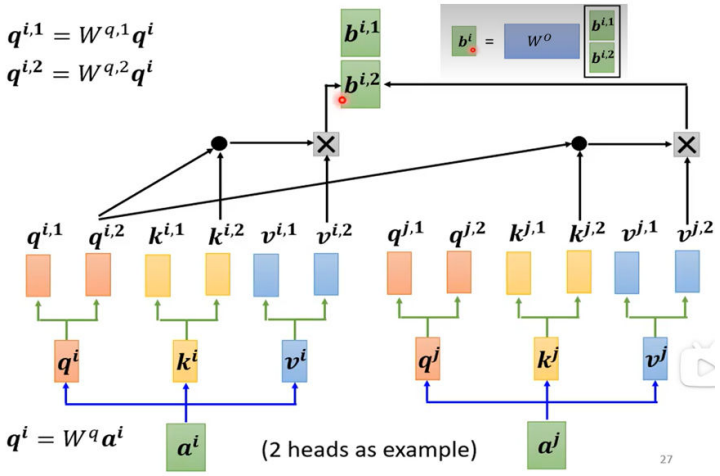
but we will use the former one in this chapter. We often refer to $\boldsymbol{q}$ as *query*, $\boldsymbol{k}$ *key*, and α *attention score*. Specially, we also calculate $\alpha_{i,i}$ in practice. Then, Softmax (or other activation function such as ReLU) is applied to $\alpha_{i,j}(j = 1,2,\ldots,n)$ for normalization and get $\alpha'_{i,j}$. Finally, we calculate $\boldsymbol{v}^j = \boldsymbol{W}^v\boldsymbol{a}^j$, and the output corresponding to $\boldsymbol{a}_i$ is

$$\boldsymbol{b}_i = \sum_{j=1}^{n} \alpha'_{\{i,j\}}\boldsymbol{v}^j$$

It is obvious that the calculation of $\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n$ is parallel, and the steps above can also be denoted as

$$\boldsymbol{Q} = \boldsymbol{W}^q\boldsymbol{I}$$
$$\boldsymbol{K} = \boldsymbol{W}^k\boldsymbol{I}$$
$$\boldsymbol{V} = \boldsymbol{W}^v\boldsymbol{I}$$
$$\boldsymbol{A} = \boldsymbol{K}^T\boldsymbol{Q}$$
$$\boldsymbol{A}' = \boldsymbol{\sigma}(\boldsymbol{A})$$
$$\boldsymbol{O} = \boldsymbol{V}\boldsymbol{A}'^{\boldsymbol{T}}$$

where $\boldsymbol{Q} = [\boldsymbol{q}_1\,\boldsymbol{q}_2\ldots\boldsymbol{q}_n]$, $\boldsymbol{K}$, $\boldsymbol{V}$ similarly, $\boldsymbol{I} = [\boldsymbol{a}_1\,\boldsymbol{a}_2\ldots\boldsymbol{a}_n]$, $A_{ij} = \alpha_{i,j}$, and $\boldsymbol{O} = [\boldsymbol{b}_1\,\boldsymbol{b}_2\ldots\boldsymbol{b}_n]$. To detect different types of relevance, we can introduce *multi-head self-attention*. We multiply each (original) $\boldsymbol{q}$, $\boldsymbol{k}$ and $\boldsymbol{v}$ by another some matrices to get some groups of (new) $\boldsymbol{q}$, $\boldsymbol{k}$ and $\boldsymbol{v}$, and calculate some output vectors as before. Then we concatenate these vectors and multiply the result by a metric to get the final output $\boldsymbol{b}$, as demonstrated below.



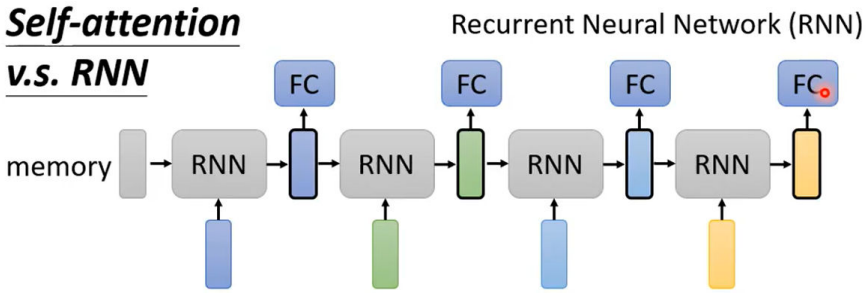
There is no position information in usual self-attention. When such information is important, we may design a unique positional vector $\boldsymbol{e}^i$ for each position and add it to the original $\boldsymbol{a}^i$. Such positional vector is usually hand-crafted but may also learned by machine, c.f. https://arxiv.org/abs/2003.09229.

Self-attention is widely used in *natural language processing* (NLP), e.g., *transformer* (c.f. https://arxiv.org/abs/1706.03762) and *BERT* (c.f. https://arxiv.org/abs/1810.04805). It can also be used for *speech recognition*, but too long an input sequence may lead to difficulty. *Truncated self-attention* can help solve the problem, c.f. https://arxiv.org/abs/1910.12997. Moreover, we

can use self-attention for graph and only attend to connected nodes, which is one type of *graph neural network* (GNN).

Self-attention can be used for image if we consider each pixel (several channels) a vector, e.g., self-attention GAN (c.f. https://arxiv.org/abs/1805.08318) and *detection transformer* (DETR, c.f. https://arxiv.org/abs/2005.12872). In fact, we can consider CNN as a simplified version of self-attention (as it only attends in a receptive field), and self-attention a complex version of CNN (as it has a learnable receptive field). To learn more about the relationship between CNN and self-attention, c.f. https://arxiv.org/abs/1911.03584. We can see that self-attention is more flexible than CNN, so it works worse than CNN with less data because of overfitting, but better with more data. It is also possible to combine these two, e.g., *conformer*.

Recurrent neural network (RNN) is another kind of network architecture that deals with an input sequence. To put it simply, RNN takes one vector from the input sequence, as well as another vector that RNN outputs when dealing with the last one input vector, as demonstrated below.



It's obvious that such RNN does not consider the vectors input later, but even bidirectional RNN behaves differently from self-attention, as it's hard for RNN to consider a vector "far from" the current position but no difference for self-attention. To learn more about the relationship between RNN and self-attention, c.f. https://arxiv.org/abs/2006.16236. Moreover, the calculation in RNN is non-parallel, which may cause more and more use of self-attention instead of RNN.

There are some researches on improving the efficiency of self-attention, c.f.
Long Range Arena: a Benchmark for Efficient Transformers, https://arxiv.org/abs/2011.04006
Efficient Transformers: a Survey, https://arxiv.org/abs/2009.06732.

Chapter 5 Transformer

Transformer is a *sequence-to-sequence* (seq2seq) model, which determines the output length itself.
Transformer has a wide use in *natural language processing* (NLP), e.g., *speech recognition* (c.f. https://sites.google.com/speech.ntut.edu.tw/fsw/home/challenge-2020), *text-to-speech* (TTS) synthesis, and chat-bot. There are also situations that can be considered as question-answering (QA), which can be done by seq2seq, while specialized models may work better. *Syntactic parsing* can be considered as translation and thus done by seq2seq (c.f. https://arxiv.org/abs/1412.7449).

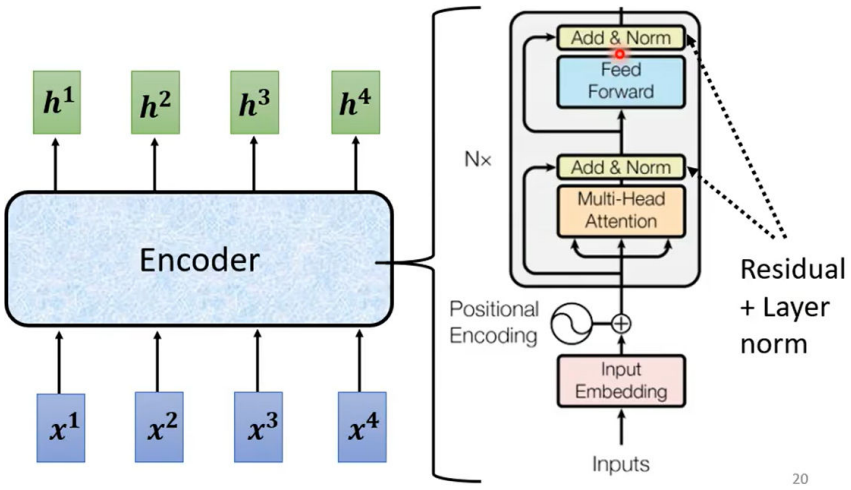
Multi-label classification (c.f. https://arxiv.org/abs/1909.03434 & https://arxiv.org/abs/1707.05495) and *object detection* (c.f. https://arxiv.org/abs/2005.12872). Transformer is composed of *encoder* and *decoder*.

Encoder

We may use RNN or CNN as encoder, but transformer's encoder is usually self-attention (with positional encoding), as demonstrated below.

Here "add" refers to *residual* connection, i.e., the final output is the sum of the output of usual self-attention and the corresponding input. "Norm" refers to layer norm, which means normalizing the vector by the mean and standard deviation of its dimensions, instead of batch norm, which means normalizing the vector by the mean and standard deviation of the same dimension of all features (i.e., examples). There is similar "add & norm" step after fully-connected network block.

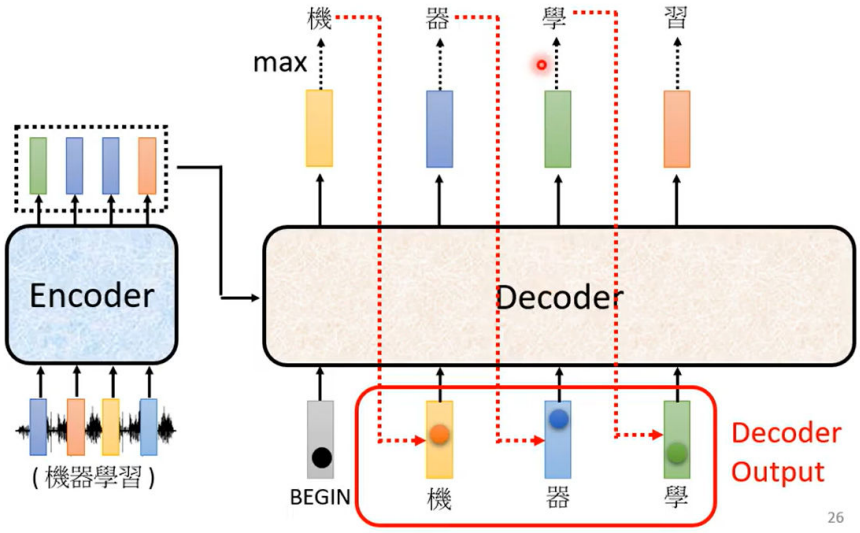
BERT use the same network architecture as transformer's encoder.



Decoder

Autoregressive (AT) model is the most commonly-used decoder. We will take speech recognition as an example.

The input sequence of AT decoder is the tokens that decoder has already output. (At the beginning is a special token referred to as *BEGIN* or *begin of sentence*, BOS.) The length of an output vector equals to the size of the vocabulary (The elements of a vocabulary may be Chinese characters, letters or sub-words.), and each element of the vector represents the probability of corresponding token. The token of the maximum probability will be the output token. The process is as demonstrated below.



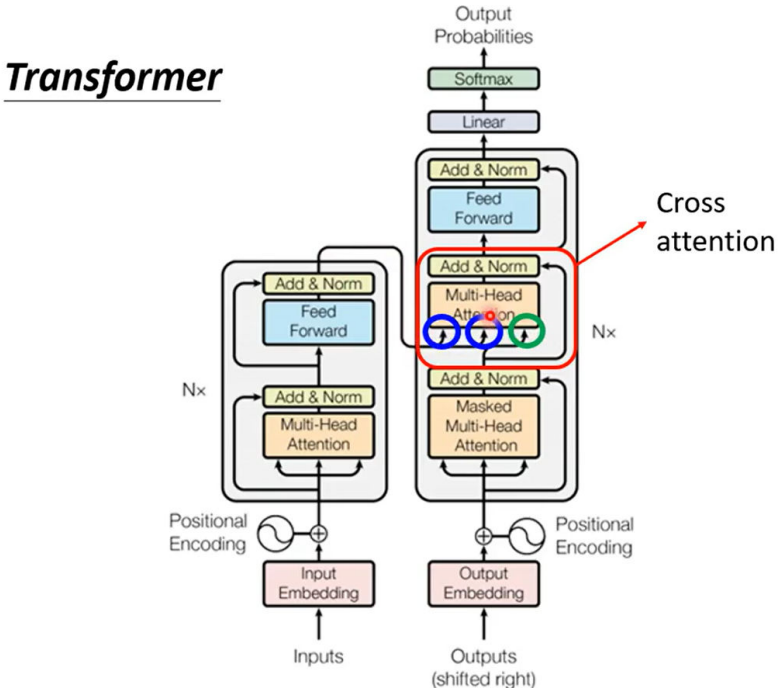
Self-attention in the decoder is a little different, referred to as *masked self-attention*. “Masked” means that the self-attention does not consider later input vectors. In fact, later input vectors are unknown, because the tokens are output one by one and serve as input.

The decoder must decide the length of output itself, so we introduce another special token into the vocabulary, referred to as *END*. BEGIN and END can, in fact, be the same token, as BEGIN only exists in input and END in output.

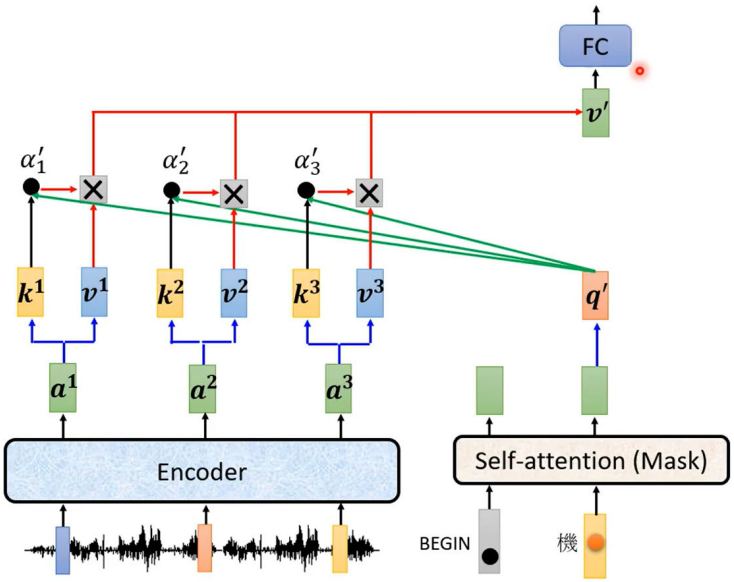
Non-autoregressive (NAT) model is another kind of decoder. The input of NAT decoder is just a sequence of BEGIN, and the decoder produces all output tokens at the same time. The length of output equals to that of the BEGIN sequence, and there are two main methods to decide it. We may train another predictor for output length, or set a length long enough and ignore the tokens after END. It is obvious that NAT decoder works parallelly based on self-attention, and its output length is controllable. However, NAT usually performs worse than AT, which may relate to multi-modality.

Encoder-decoder

The part of network that links encoder and decoder is referred to as *cross-attention*, as demonstrated below.



Cross-attention here is similar to masked self-attention, but the query vectors ($\boldsymbol{q}$) are calculated from the input tokens and the key vectors ($\boldsymbol{k}$) and value vectors ($\boldsymbol{v}$) are calculated from the output sequence of encoder, as demonstrated below. Decoder thus get information from encoder.



Conventional cross-attention takes the output of the last layer of encoder. However, as there are many layers in encoder and decoder, it is not necessarily the case, c.f. https://arxiv.org/abs/2005.08081.

How to Train

The ground truth is a one-hot vector and the output of transformer is a distribution vector, so we minimize the cross-entropy of them to train the model. We can see that the generation of a token

is similar to classification. We use the ground truth as the input tokens of decoder when training (instead of those generated by itself), which is referred to as *teacher forcing*.

One tip for training is called copy mechanism. There are cases that some of the output is copied from the input, e.g., chat-bot and summarization (c.f. https://arxiv.org/abs/1704.04368). *Pointer network* and *copy network* (c.f. https://arxiv.org/abs/1603.06393) can do that.

Another tip is guided attention. It introduces certain limitations on the attention scores' distribution, which is meaningful in tasks where input and output are monotonically aligned, e.g., speech recognition and TTS. (For example, we know in TTS the model should attend more on the 2nd input token when generating the 2nd output, so we may apply such limitation to the model when training.) Monotonic attention and location-aware attention can do that.

The 3rd tip called *beam search*. The model chooses the token with the most probability, which is called *greedy decoding*. However, such a sequence (or we can say "path") is not necessarily the best one, and it is impossible to check all paths. Beam search gives an approximate solution to this problem, but it does not always work.

The 4th tip is sampling. Beam search may help in tasks with one certain answer, but decoder needs randomness to work better on tasks with open answers like TTS and sentence completion, c.f. https://arxiv.org/abs/1904.09751. For randomness, one method is to add noise to the input when testing.

We compare the output of model and ground truth (both a sequence) and use *BiLingual Evaluation Understudy* (BLEU) *score*, instead of the cross-entropy between each corresponding token, to evaluate the model's performance. Therefore, we use BLEU score for validation. We cannot use BLEU score for training as it is not differentiable, but we can still use it as the reward for *reinforcement learning*, c.f. https://arxiv.org/abs/1511.06732.

The input of decoder is ground truth when training while the output of itself when testing, and such a mismatch is referred to as *exposure bias*. We may use *scheduled sampling* to help solve this problem, which means input something wrong to the decoder. However, scheduled sampling may affect the parallel calculation of self-attention, c.f. https://arxiv.org/abs/1506.03099, https://arxiv.org/abs/1906.07651, https://arxiv.org/abs/1906.04331.

To learn more

The encoder described earlier is the original version, and there are some later and more optimal versions:

On Layer Normalization in the Transformer Architecture, https://arxiv.org/abs/2002.04745

PowerNorm: Rethinking Batch Normalization in Transformers, https://arxiv.org/abs/2003.07845

Chapter 6 Generative Model

Introduction of Generative Models

Network as *generator* takes not only usual x (scalar, vector or sequence) but also z sampled from a certain distribution as input, and generate y (scalar, vector or sequence) as output. Therefore, we must know the formulation of the distribution so that we can sample from it, and the same input x can lead to different output y as z is different. The step of sampling is important for the tasks that need "creativity", because the same x can have different ground truth and the model may try to meet them at the same time (instead of meet one at a time) without sampling.
Generative adversarial network (GAN) is one of the most famous generative models.

Generative Adversarial Network (GAN)

Basic Idea

We will discuss unconditional generation first, and take image generation as an example.
In unconditional generation, there is no x and z is usually a low-dim vector sampled from normal distribution (or other distribution, but may not matter), and the output is usually a high-dim vector (image is vector).
Discriminator is another network which output a scalar, other than generator. A larger value of the scalar means the input of discriminator is real, while smaller value means false.
With generator and discriminator, the basic idea of GAN is as following. The generator generates some images (fake images), and the discriminator learns to classify real and fake images. Then the generator learns to "fool" the discriminator, and this process is repeated several times.

Algorithm

The algorithm of GAN is stated as following. We initialize a generator and a discriminator. Then in each training iteration, there are two steps. Step 1, we fix the generator and update the discriminator - discriminator learns to assign high scores to real objects (sampled from database) and low scores to generated ones, which can be considered as classification or regression. Step 2, we fix the discriminator and update the generator – we connect the generator and discriminator together as a large network, fix the layers of discriminator and maximize the large network's output.
Theoretically, discriminator should be trained from scratch every time, but we usually use the parameters of the discriminator last time in practice.

Some Famous GANs

Progressive GAN (c.f. https://arxiv.org/abs/1710.10196) can generate HD human face images. As the input of generator is just a vector, we may do interpolation to get new, compromise images.
BigGAN (c.f. https://arxiv.org/abs/1809.11096) can generate many different images, although some of them are imperfect.

Theory behind GAN

The generator G transfers normal distribution (from which we sample z) to another distribution P_G. Denote the real data's distribution as P_{data}, we expect P_G to be as close to P_{data} as possible, i.e., our objective is

$$G^* = arg\ \min_G Div(P_G, P_{data})$$

where $Div(P_G, P_{data})$ is the divergence between P_G and P_{data}.
It is difficult to compute the divergence. Fortunately, sampling is good enough, and although we do not know P_G or P_{data}, we can sample from them. In fact, we do that with discriminator: when training discriminator, our objective is

$$D^* = arg\ \max_D V(G, D)$$

where

$$V(G, D) = E_{y \sim P_{data}}[log\ D(y)] + E_{y \sim P_G}\big[log\ \big(1 - D(y)\big)\big]$$

is the objective function for D. Or, we can minimize $-V(G, D)$, which is the cross-entropy and to train D is to train a classifier. Moreover, we can see that $V(G, D)$ is related to *JS divergence*, c.f. https://arxiv.org/abs/1406.2661.
Therefore, we can see that

$$G^* = arg\ \min_G \max_D V(G, D)$$

and the algorithm stated earlier aims to solve this.
We can use other kinds of divergence as well, c.f. https://arxiv.org/abs/1606.00709.

Tips for GAN

WGAN

As the saying goes, no pain, no GAN. It is quite difficult to train GAN.
P_G and P_{data} are not overlapped in most cases for two reasons. First, both P_G and P_{data} are low-dim manifolds in high-dim space and their overlap can be ignored. Second, we never know these two distributions but simply sample from them, so even though P_G and P_{data} have significant overlap, we will not see it without enough samples. This fact causes a problem, as the algorithm stated earlier aims to minimize the JS divergence of P_G and P_{data}, but the JS divergence of two distributions without overlap is always $log\,2$. (Intuitively, binary classifier

achieves 100% accuracy if two distributions do not overlap, thus the accuracy, or loss, is meaningless during training.)

Therefore, we may use *Wasserstein distance* (here 'w' is pronounced as 'v') instead of JS divergence. We can imagine P_G and P_{data} as two piles of earth, and Wasserstein distance of them $W(P_G, P_{data})$ is defined as the smallest average moving distance to move P_G to P_{data}. Now we can see that $W(P_G, P_{data})$ decreases as P_G and P_{data} become closer, i.e., generator becomes better.

GAN using Wasserstein distance is referred to as *WGAN* (c.f. https://arxiv.org/abs/1701.97875). It is given without proof that

$$D^* = arg\ \max_{D \in 1-Lipschitz} \left(E_{y \sim P_{data}}[D(y)] - E_{y \sim P_G}[D(y)]\right)$$

Here

$$\max_{D \in 1-Lipschitz} \left(E_{y \sim P_{data}}[D(y)] - E_{y \sim P_G}[D(y)]\right)$$

is just the Wasserstein distance between P_G and P_{data}. Discriminator D should be smooth enough, as the training of D will not converge without that constraint. To meet the constraint, original WGAN forces the weight parameters to be bounded (if $w > c$, let $w = c$; if $w < -c$, let $w = -c$). We can also use *gradient penalty* (c.f. https://arxiv.org/abs/1704.00028) or *spectral normalization* (such GAN called SNGAN, c.f. https://arxiv.org/abs/1802.05957).

Even with these tips, GAN is still challenging, as when one of generator and discriminator fails to improve, the other cannot improve either.

Here are some more tips: tips from Soumith, c.f. https://github.com/soumith/ganhacks; tips in DCGAN (guideline for network architecture design for image generation), c.f. https://arxiv.org/abs/1511.06434; improved techniques for training GANs, c.f. https://arxiv.org/abs/1606.03498; tips from BigGAN, c.f. https://arxiv.org/abs/1809.11096.

GAN for Sequence Generation

We can consider the decoder as generator and use GAN for sequence generation. However, the decoder outputs the tokens with maximum probability and a small change in decoder may not change the output tokens, making it nondifferentiable. Therefore, we have to train it with RL.

We know that both RL and GAN are difficult to train, so it is extremely difficult to train such GANs. Usually, the generator is fine-tuned from a model learned by other approaches. Yet with enough hyperparameter-tuning and tips, it is still possible to train such GANs from scratch, c.f. https://arxiv.org/abs/1905.09922.

Other Generative Models

There are other generative models like *variational autoencoder* (VAE) and *FLOW-based model*, but GAN is usually better.

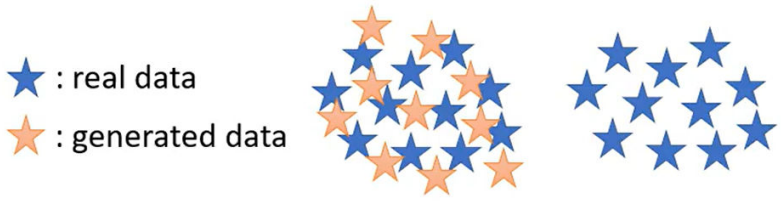
We can also pair vectors sampled from Gaussian distribution and real objects, and then train a generative model with typical (supervised) learning approaches. However, randomly-paired vectors and objects may not behave well, and we should apply some other tips. *Generative latent optimization* (GLO, c.f. https://arxiv.org/abs/1707.05776) and *gradient origin networks* (c.f. https://arxiv.org/abs/2007.02798) are such models.

Evaluation of Generation

It is not easy to evaluate the generation. Human evaluation is expensive, and sometimes unfair and unstable.

One method to evaluate images automatically is as following: we input a generated image y^j to an off-the-shelf image classifier (e.g., inception net, VGG, etc.) and get the probability that the image belongs to one certain class c_i $P(c_i|y^j)$. A concentrated distribution of $P(c_i|y^j)$ means higher visual quality.

However, the generated data may concentrate near one or a few samples from real data and has little diversity, which is referred to as *mode collapse*. Mode collapse is nearly unavoidable (although we can stop the training earlier before it happens). Moreover, the generated data can concentrate near one part of real data and lose some diversity, which is referred to as *mode dropping* and is even harder to avoid, as demonstrated below.



Therefore, we should not only consider the distribution of $P(c_i|y^j)$ for each y^j, but also consider the distribution of $P(c_i) = \frac{1}{N}\sum_{j=1}^{N} P(c_i|y^j)$ for each c_i. A uniform distribution of $P(c_i)$ means higher variety.

As quality and diversity conflicts to some extent, we should consider them both in evaluation. One way is to use *inception score* (IS) based on *inception net*, which becomes larger with high quality and high diversity. But as similar (but still diverse) images may be classified as one class, it is better to use *Fréchet inception distance* (FID), which use the input of the Softmax layer, instead of the output, to represent an image, c.f. https://arxiv.org/abs/1706.08500. FID considers the Fréchet distance between two Gaussian distributions, so a smaller FID is better. FID needs a lot of samples, and the assumption that the distribution is a Gaussian can cause problems as well, so it may be wiser to consider more than one index.

There are more problems, e.g., we do not want "memory GAN" that generates the same images as real data, which can achieve low FID but is meaningless. To learn more about evaluation, c.f. https://arxiv.org/abs/1802.03446.

Conditional Generation

As explained earlier, what we have talked about is unconditional generation. Now we are going to discuss *conditional generation* in which the input of generator is not only a random vector z but also a specified x. For example, we can use conditional generation for text-to-image tasks (c.f. https://arxiv.org/abs/1605.05396) and x is the text, and we can get different images with the

same text. Obviously, we need data with labels to train such generators.

The discriminator described earlier is not suitable for conditional generation, because it will make the generator generate realistic images but completely ignore the condition x. Therefore, discriminator of conditional GAN should take not only generator's output y, but also x as input, and the output scalar will be large when y is realistic as well as matched with x. That is to say, the data used is true image-text pairs and real images with unmatched text, and we expect the discriminator to output 1 for true image-text pairs and 0 for unmatched ones and generated ones.

Conditional GAN can be used for image translation (a.k.a. pix2pix) tasks as well, c.f. https://arxiv.org/abs/1611.07004. In such tasks, GAN usually performs better than supervised learning, and GAN along with supervised learning performs more better, where the generator tries not only to fool the discriminator but also to generate images as close to examples as possible. Conditional GAN can be used even for sound-to-image tasks (c.f. https://arxiv.org/abs/1808.04108) and "talking head generation" (c.f. https://arxiv.org/abs/1905.08233).

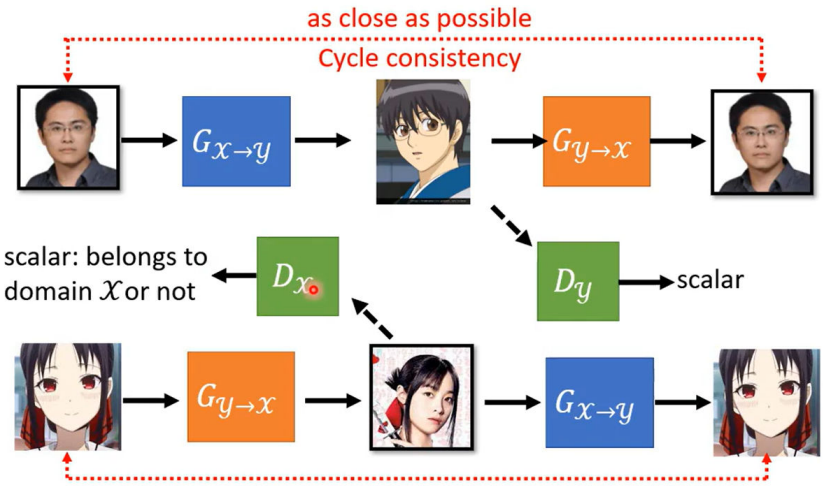
Learning from Unpaired Data

To train a network that take the input x and output y, we usually use (x, y) pairs, and this method is referred to as *supervised learning*. However, we not always have paired data and have to train with unpaired data, which is referred to as *unsupervised learning*. *Pseudo labelling* and *back translation* can deal with cases that we have a little paired data and more unpaired data. (Such method is also referred to as *semi-supervised learning*.) However, there are tasks that we can obtain completely no paired data, e.g., image style transferring (to transfer images from one style to another).

To solve problems with no paired data, we may turn to unsupervised conditional generation with GAN. That is to say, we expect the generator $G_{x\to y}$ to take images belonging to $Domain\ X$ as input and generate images belonging to $Domain\ Y$, and the discriminator D_y to output a scalar whose value means its input belongs to $Domain\ Y$ or not.

There is still one problem: generator trained this way may ignore the input image, as what we have discussed about conditional GAN, but we have no paired data to train GAN as before. So we introduce *Cycle GAN* (c.f. https://arxiv.org/abs/1703.10593). There is another generator $G_{y\to x}$ in cycle GAN which takes the output of $G_{x\to y}$ as input and generates another image. In addition to usual GAN's training targets, we also expect $G_{y\to x}$'s output as close to $G_{x\to y}$'s input as possible, which is called *cycle consistency*. (Here "close" means a small distance between the two vectors representing the two images.) We can see that $G_{x\to y}$'s output has to "relate to" its input so that it is possible for $G_{y\to x}$ to reconstruct the image. Cycle consistency may not work sometimes and there is no good way to avoid that yet, but fortunately, it works in most cases.

Cycle GAN can also be bidirectional. We can train from the other direction as well, i.e., exchange the position of $Domain\ X$ and $Domain\ Y$ and train $G_{y\to x}$ and $G_{y\to x}$ (here we need another discriminator D_x), as demonstrated below.



There are other kinds of GANs suitable for image style transferring, e.g., *Disco GAN* (c.f. https://arxiv.org/abs/1703.05192) and *Dual GAN* (c.f. https://arxiv.org/abs/1704.02510). Their idea is similar to cycle GAN. *Star GAN* (c.f. https://arxiv.org/abs/1711.09020) is an advanced version of cycle GAN. For more about advanced cycle GAN, c.f. https://arxiv.org/abs/1907.10830.

Such methods can be applied to other tasks like *text style transferring* (e.g., transfer negative text to positive). As the outputs are texts, there is some difference: we may use transformer, the method to measure how close two texts are is a little complex, and the (non-differentiable) discriminator has to be trained by RL.

We can also use such methods for unsupervised abstractive summarization (c.f. https://arxiv.org/abs/1810.02851), unsupervised translation (c.f. https://arxiv.org/abs/1710.04087 and https://arxiv.org/abs/1710.11041) and unsupervised ASR (automatic speech recognition, c.f. https://arxiv.org/abs/1804.00316, https://arxiv.org/abs/1812.09323 and https://arxiv.org/abs/1904.04100).

Chapter 7　Self-Supervised Learning

Sesame Street and 進撃の巨人

Many self-supervised learning models are named after characters in Sesame Street, such as ELMo for *embeddings from language models*, BERT for *bidirectional encoder representations from transformers*, ERNIE for *enhanced representation through knowledge integration* and Big Bird for *transformers for longer sequences*.

BERT is a large model with 340 million parameters. The controller of Colossal Titan in 進撃の巨人, Bertolt Hoover, also has “bert” in his name. It can’t be a coincidence!

Large models nowadays have had more than 1T parameters. We will talk about BERT series and GPT series in this chapter.

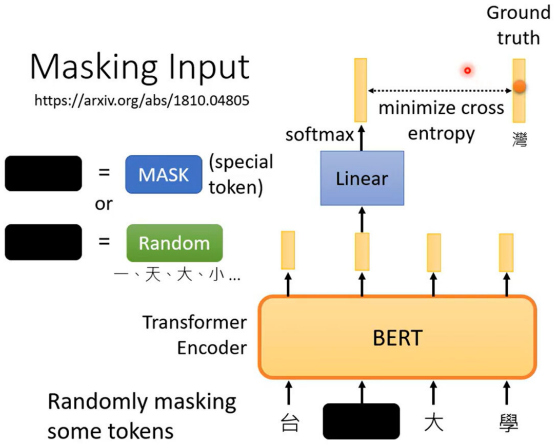
BERT Series

Self-Supervised Learning

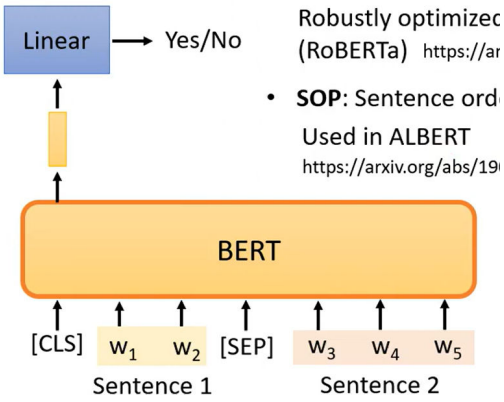
We have discussed supervised learning before. If we have no label $\hat{y}$s for data xs, we may choose one part of x, x' as the model's input and the other part x'' as label. Such a method is referred to as *self-supervised learning*, and can also be considered as one kind of unsupervised learning.

Masking Input, Next Sentence Prediction

BERT, a transformer encoder, does self-supervised learning by masking input, c.f. https://arxiv.org/abs/1810.04805. BERT, obviously, takes a sequence of vectors as input, and we will take the token sequence as an example. BERT randomly masks some of the tokens, and to mask a token, we replace it either with a special token *MASK* or another token randomly selected from the vocabulary. Then, the output vector corresponding to the masked token (MASK or a random one) "goes through" a linear layer and Softmax, thus becomes a vector whose elements are probability of tokens. The token under the mask is the ground truth and represented by a one-hot vector, and we use cross-entropy as loss function, as demonstrated below. (We can also consider it as a classification task whose number of classes equals to that of all tokens.)



In *next sentence prediction*, BERT is trained to predict whether two sentences are linked or not. To be specific, the input of BERT is a special token *CLS* at the beginning, followed by two sentences separated by another special token *SEP*, and the output of BERT corresponding to CLS is expected to represent the prediction ("yes" or "no") after "going through" a linear layer, as demonstrated below. However, some research found it unhelpful, c.f. https://arxiv.org/abs/1907.11692. The reason why it does not help may be that such a task is too easy, and *sentence order prediction* seems helpful (c.f. https://arxiv.org/abs/1909.11942).



Pre-Train and Fine-Tune

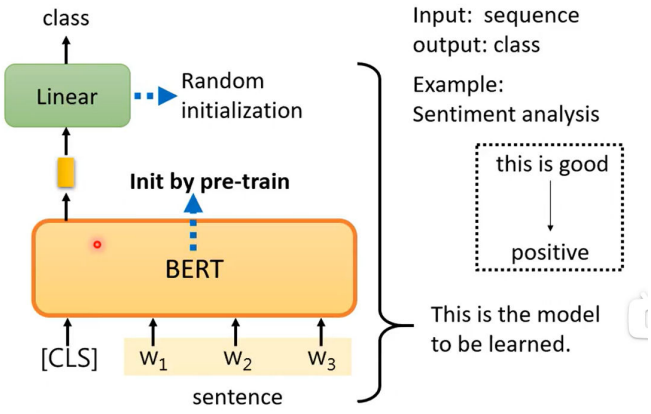
From what we have talked about, we can see that BERT is trained to fill the blanks, but magically, it can do tasks that we care, referred to as *downstream tasks*, even though they have no clear relationship with blank-filling. To do downstream tasks, we need to get a BERT first by self-supervised learning, referred to as *pre-train*, and then *fine-tune* it with a little bit labelled data. Therefore, the whole process is semi-supervised, as the pre-train is unsupervised and fine-tune supervised, and it usually performs better than models trained from scratch.

Before introducing how to fine-tune BERT, it is important to evaluate a BERT. Naturally, we should test BERT on a set of tasks, and one famous set is *General Language Understanding Evaluation* (GLUE, c.f. https://bluebenchmark.com/ and its Chinese version https://www.cluebenchmarks.com/).

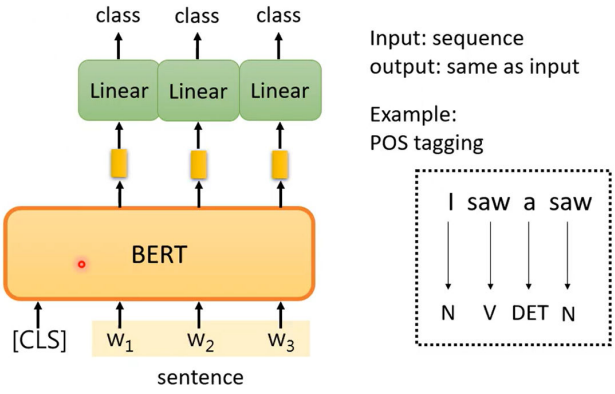
How to Use BERT

Here are some examples on how to use BERT.

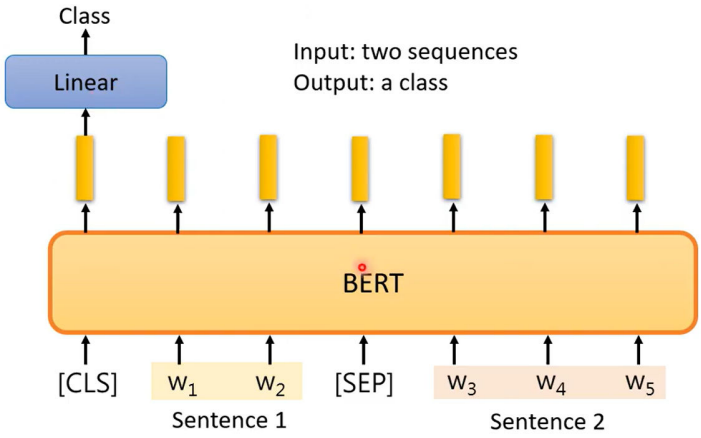
Case 1, one sequence as input and one class as output, sentiment analysis as an example. The input sequence of BERT is CLS followed by the sentence, and the output of BERT corresponding to CLS "goes through" a linear layer and finally predict whether the sentence is positive or negative. When training this model with the labelled data we have, we update parameters in both BERT and the linear layer, but the linear layer is initialized randomly while BERT initialized by the parameters of the pre-trained BERT. The whole process is demonstrated below.



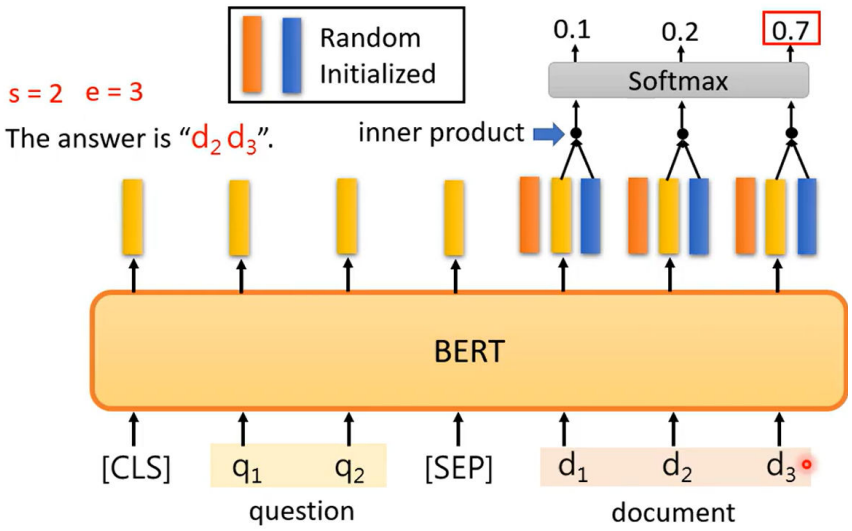
Case 2, one sequence as input and another sequence of the same length as output, POS tagging as an example. It is similar to case 1, as demonstrated below.



Case 3, two sequences as input and one class as output, *natural language inference* (NLI, one input sentence is a premise and the other a hypothesis, and the output is "neutral", "entailment" or "contradiction") as an example. It is as demonstrated below.



Case 4, extraction-based question answering (QA), one kind of QA tasks where all answers are supposed to exist in the document. To be specific, the model takes the input of the document $D = \{d_1, d_2, ..., d_n\}$ and the query $Q = \{q_1, q_2, ..., q_m\}$, and outputs two integers s and e representing the answer $A = \{d_s, d_{s+1}, ..., d_e\}$. The model includes a BERT and two (randomly initialized) vectors of the same length as the output vectors of BERT. One of these two vectors does inner product with the output vectors of BERT corresponding to the document's vectors (d_i), and the products are then applied with Softmax to get a probability distribution, and the document's vector corresponding to the highest probability is d_s. The model gets d_e similarly with the other vector. The process is as demonstrated below. The length of BERT's input sequence cannot be too long (usually shorter than 512 tokens), as BERT is a transformer encoder and needs large amounts of computation, and we usually cut the input document into pieces.



As the pre-train of BERT is costing, there are researches on when BERT "gets certain ability" during training, a.k.a. BERT embryology, c.f. https://arxiv.org/abs/2010.02480.

We can see that the four cases above do not encompass seq2seq cases. To pre-train a seq2seq model is different, as we need to train an encoder as well as a decoder linked by cross-attention. When pre-training, we corrupt the input tokens by one or some of the following methods: masking, deleting typical letters (e.g., deleting all 'D'), permutation and rotation (c.f. https://arxiv.org/abs/1905.02450 and https://arxiv.org/abs/1910.13461, the paper *Transfer Text-to-Text Transformer* compares these methods), and expect the decoder's output to reconstruct the input (i.e., output the same token sequences as the input).

Why BERT works

An output vector of BERT, a.k.a. *embedding*, represents the meaning of a token. That is to say, the tokens with similar meanings have similar embeddings, and one same token can have more than one embedding as the context differs (We can prove that by computing the cosine similarities between the embeddings of the same token with different contexts.). Many people believe that BERT learns the meaning of tokens by the company it keeps, and BERT's embedding is referred to as contextualized word embedding.

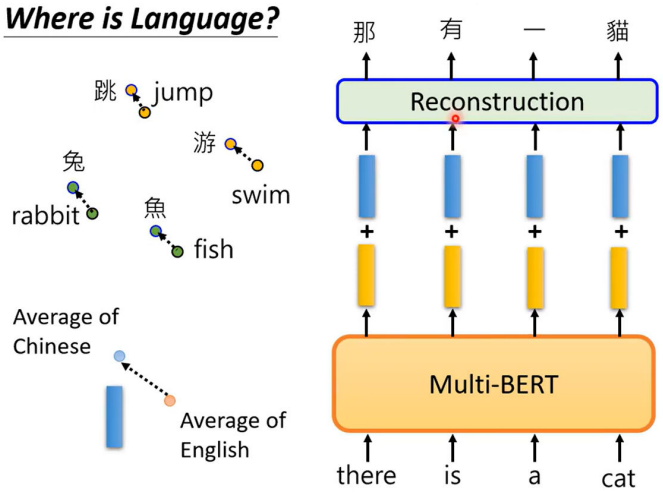
This explanation may not be the complete truth. It is found that BERT pre-trained by words can do DNA (represent four kinds of deoxynucleotide with four words), protein or music classifications, and performs better than randomly initialized models (c.f. https://arxiv.org/abs/2103.07162).

Multi-Lingual BERT

Multi-lingual BERT is a BERT pre-trained with multiple languages. When testing with one certain language, multi-lingual BERT fine-tuned by another language performs just a little worse than the one fine-tuned by the same language. Therefore, some people believe that the embeddings of similar meanings are similar, regardless of the language. However, it takes a lot of data and time

to train such a BERT, or it will not perform better than a one-language one.

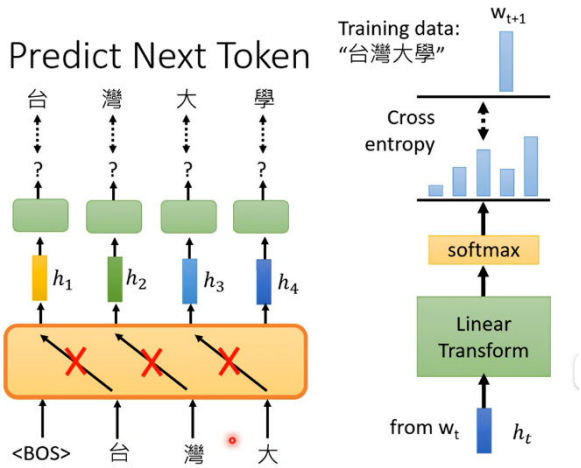
We can see that language information still exists, as it will be impossible to reconstruct the input correctly if embeddings are independent of language. In fact, the difference between the average of one language's embeddings and that of another language's embeddings represents the language information. If we add the difference to one language' embeddings, we will get the other's, as demonstrated below.



GPT Series

Pre-Train

Different from BERT, GPT is trained to predict next token (the first input is BEGIN to predict the first token), as demonstrated below. The model here is, in fact, a decoder except cross-attention, i.e., the self-attention in the model is masked.



GPT can thus do generation, and its image is usually a unicorn as one GPT generated a famous (fake) report on a unicorn.

How to Use GPT

We can connect GPT to a linear classifier, as what we have done on BERT. However, GPT is usually used in a different way, partly because it is so large (so many parameters) that even fine-tune is very difficult.
We usually use GPT by giving it task description, a few examples and a prompt, without updating it with gradient descent. Such a method is referred to as *few-shot learning*. There are also another two similar methods: *one-shot learning* with just one example and *zero-shot learning* with no example. These three methods are also referred to as *in-context learning*.

Beyond Text

Self-supervised learning can be used for not only NLP, but also speech and *computer vision* (CV).
SimCLR is a famous usage of self-supervised learning for image, c.f. https://arxiv.org/abs/2002.05709 and https://github.com/google-research/simclr. BYOL, another usage for image, seems impossible but works, c.f. https://arxiv.org/abs/2006.07733. We can transfer audio to a sequence of vector, so we can train BERT or GPT for speech in a similar way to that for text. (A model for speech needs to know how to process content, speaker, emotion, and even semantics.)

Chapter 8 Auto-Encoder

Basic Idea of Auto-Encoder

We have seen that the model needs to do self-supervised learning, or be pre-trained, to do downstream tasks when most of data we have is unlabeled. We have talked about some methods in the last chapter, and *auto-encoder* is another method of self-supervised learning (although its idea came out much earlier than the concept of self-supervised learning).
Auto-encoder consists of two neural network, encoder and decoder. Encoder takes the input (an image or something else) and outputs a vector, and decoder takes this vector as input and is expected to reconstruct the encoder's input (i.e., we expect the two vectors representing encoder's input and decoder's output to be as close as possible). We can see that this process is similar to cycle GAN, and is unsupervised. The vector encoder outputs is referred to as embedding, a.k.a. *representatio*n or *code*, and can be used as new features for downstream tasks.
As the input of encoder (the original feature) is usually high-dim while embedding (the new feature) low-dim, encoder actually does *dimension reduction*, and the corresponding part of network is also called "bottleneck". (*PCA* and *t-SNE* are another two methods of dimension reduction not based on deep learning.)

Why Auto-Encoder works

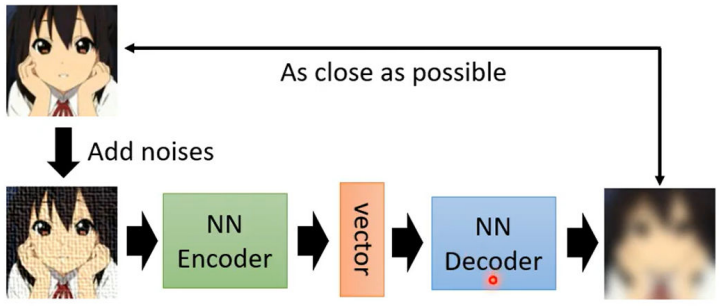
Although the original feature is high-dim, not all information is important. (Only a small set of all 4D tensors are images and the other are meaningless, obviously.) That may be the reason why auto-encoder works.

Auto-Encoder Is Not a New Idea

Auto-encoder is not a new idea, dating back to 2006. Auto-encoder then was trained with the help of *restricted Boltzmann machine* (RBM), which is seldom used nowadays. Early researchers also believed that the network of encoder and decoder must be symmetric, which is found unnecessary.

De-noising Auto-Encoder

De-noising auto-encoder is a variation of auto-encoder. The input of de-noising auto-encoder' encoder is added with noise and decoder is expected to reconstruct the input without noise, as demonstrated below.



The idea sounds familiar? ☺

Vincent, Pascal, et al. "Extracting and composing robust features with denoising autoencoders." *ICML*, 2008.

BERT can be considered as a de-noising auto-encoder, masking considered as adding noise. The "decoder" in BERT is not necessarily the linear layer and "embedding" not necessarily the output of BERT (which we refer to as embedding in Chapter 7), and we can decide by ourselves that which part of it is "encoder" or "decoder".

Feature Disentanglement

Representation includes information of different aspects, e.g., an audio's representation includes information about content, speaker, etc. It is possible to tell their relationship, e.g. content information is represented by a certain part of the representation, and such process is referred to

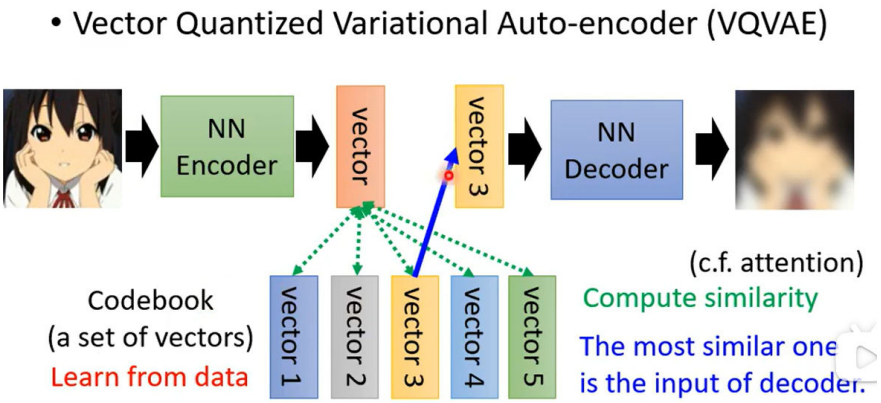
as *feature disentanglement*, c.f. https://arxiv.org/abs/1904.05742, https://arxiv.org/abs/1804.02812 and https://arxiv.org/abs/1905.05879.

Feature disentanglement can be applied to voice conversion, which means transferring the speaker of certain audio to another person. Previously, we need these two persons to say exactly the same sentences, which is costing and even impossible. With the help of feature disentanglement, however, the model may learn voice conversion with two speakers talking different things by replacing the representation corresponding to speaker information.

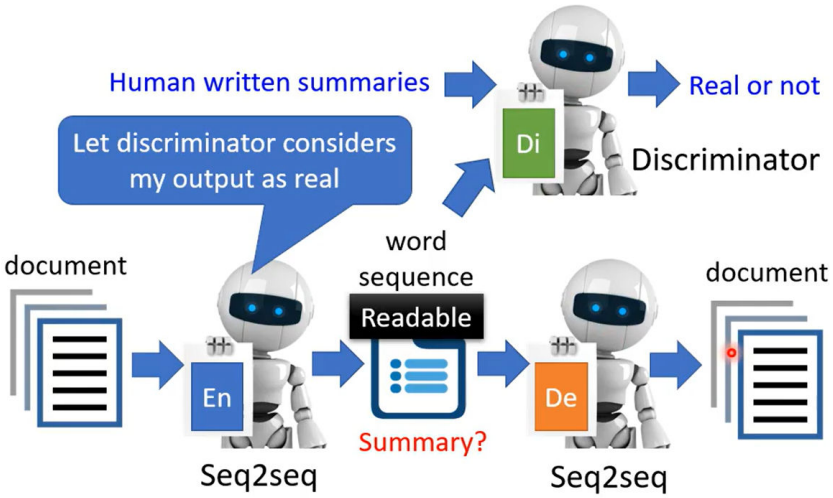
Discrete Latent Representation

The representation we discussed earlier is a vector whose elements are real numbers, but that is not necessarily the case. We may force the representation to be a binary vector whose element is either 1 or 0, and each element may represent the existence of certain feature. We can even force it to be a one-hot vector so that the model can learn unsupervised classification. Such methods are referred to as *discrete latent representation*.

Vector quantized variational auto-encoder (VQVAE, c.f. https://arxiv.org/abs/1711.00937) is a famous example of discrete latent representation. In VQVAE, we have a codebook (a set of vectors learned from data), and decoder's input is the vector most similar to encoder's input in the codebook, as demonstrated below. For speech, the codebook represents phonetic information, c.f. https://arxiv/org/pdf/1901.08810.pdf.



Even text (a word sequence) can be representation, c.f. https://arxiv.org/abs/1810.02851. Such encoder and decoder are seq2seq models, and such an auto-encoder is referred to as seq2seq2seq auto-encoder. However, such text representations are usually unreadable and for unsupervised summarization, we need to add a discriminator, as demonstrated below. Here we need RL to deal with text. In fact, such a model is no different from what we have discussed in cycle GAN.



There are also researches on tree as embedding, c.f. https://arxiv.org/abs/1806.07832 and https://arxiv.org/abs/1904.03746.

More Application

Generator

We have seen some usage of auto-encoder's encoder, and decoder is also useful. With some modification, auto-encoder's decoder can serve as a generator whose input is a vector sampled from a certain distribution. *Variational auto-encoder* (VAE) is a famous example.

Compression

Auto-encoder may also be used for compression, encoder doing compression and decoder decompression., c.f. https://arxiv.org/abs/1708.00838 and https://arxiv.org/abs/1703.00395. Such compression is lossy as the output of decoder is not exactly the same as the input of encoder.

Anomaly Detection

Auto-encoder may be used for *anomaly detection* as well. In anomaly detection, the detector is given a set of training data $\{x^1, x^2, ..., x^N\}$, and detects whether input x is similar to the training data or not. (x similar to training data is normal, while x different from training data is anomaly, a.k.a. outlier, novelty or exceptions.)

Anomaly detection can be applied to fraud detection (c.f. https://www.kaggle.com/ntnu-testimon/paysim1/home and https://www.kaggle.com/mlg-ulb/creditcardfraud/home), network intrusion detection (c.f.

http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html), and cancer detection (c.f. https://www.kaggle.com/uciml/breast-cancer-wisconsin-data/home).
Anomaly detection seems similar to binary classification, but it is difficult to find anomaly data and we usually have normal data only, i.e., we only have one class. As we train the auto-encoder with normal data, it cannot reconstruct anomaly data, and large reconstruction loss thus indicating anomaly input.

Chapter 9 Explainable Machine Learning

Sometimes we want the machine to give not only an answer but also the reason why it gives this answer. Such machine learning is referred to as *explainable machine learning*.
Explainable ML is important, as a machine giving correct answers is not necessarily intelligent. There are also cases that we must explain our model, e.g., loan issuers are required by law to explain their models, a medical diagnosis model responsible for human life cannot be a black box, a model used at the court should be non-discriminatory, a self-driving car's abnormal actions need to be explained, etc. Moreover, we can improve our model based on explanation.

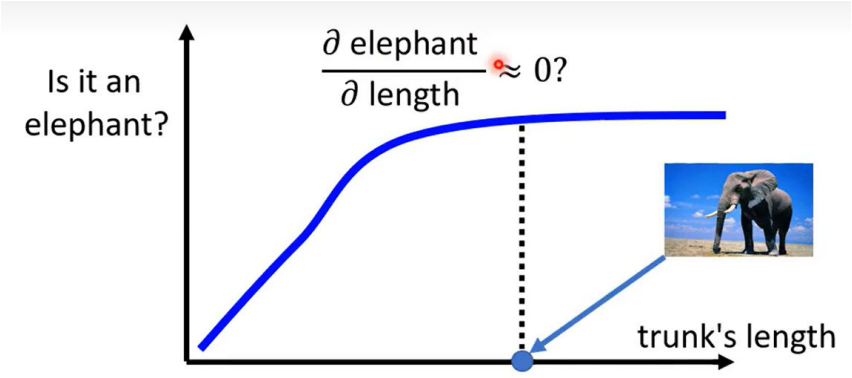
Some of the models we have today are intrinsically interpretable (E.g., we know the importance of features from the weights of a linear model.), but not very powerful. Another some, however, are difficult to interpret but more powerful. Some people think a model which is a black box should not be used even though it is more powerful, but such an opinion is no different from cutting the feet to fit the shoes.
(The word "explainable" often describes a model which used to be a black box, while "interpretable" describes a model which is not a black box originally. The difference between these two words is not always clear in usage.)
Decision tree seems both powerful and interpretable, but a tree can still be terribly complex and we actually use *random forest* in practice. We still need to study explainable ML. However, we may not need to completely know how a machine works. (We do not even know how brains work, c.f. the famous copy machine study by Ellen Langer, Harvard University. In some sense, the explanation just needs to make humans comfortable.)

There are two kinds of explanation, local explanation attending on each specific input and global explanation not.

In local explanation, we wonder in the object $x = \{x_1, x_2, \ldots, x_N\}$ which components (x_i) are critical for decision. We may remove or modify some of these components (e.g., cover a part of the input image), and if large decision change occurs, we know that the corresponding components are important. An advanced method is to compute gradient, i.e., add a small Δx to one of the components x_i, then the loss will change by a small Δe, and the value of $\left|\frac{\Delta e}{\Delta x}\right|$ (an approximation of $\left|\frac{\partial e}{\partial x_i}\right|$) indicates the importance of x_i.

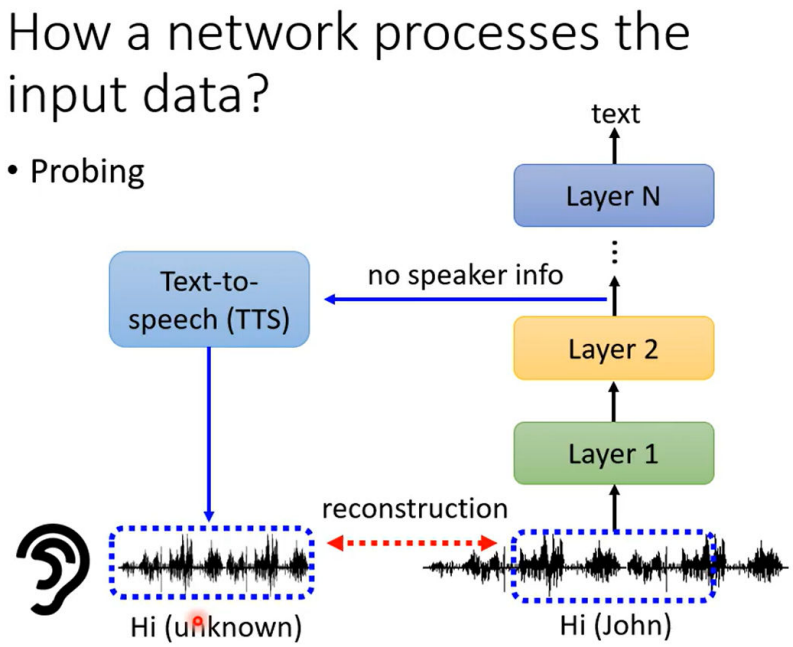
For image classification, we can draw a black-and-white image whose pixels representing the value of $\left|\frac{\Delta e}{\Delta x}\right|$ of the corresponding pixels (The larger the value is, the whiter the pixel is.), referred to as *saliency map*. Saliency map can help us find the potential problem of the model. A typical saliency map has much noise, and we may randomly add noises to the input image, get the saliency maps of the noisy images and average them, called SmoothGrad, c.f. https://arxiv.org/abs/1706.03825. However, a SmoothGrad saliency map is not necessarily better, as it may cover something important.

However, gradient does not always reflect importance because of gradient saturation, as demonstrated below. An alternative way is *integrated gradient* (IG), c.f. https://arxiv.org/abs/1611.02639.



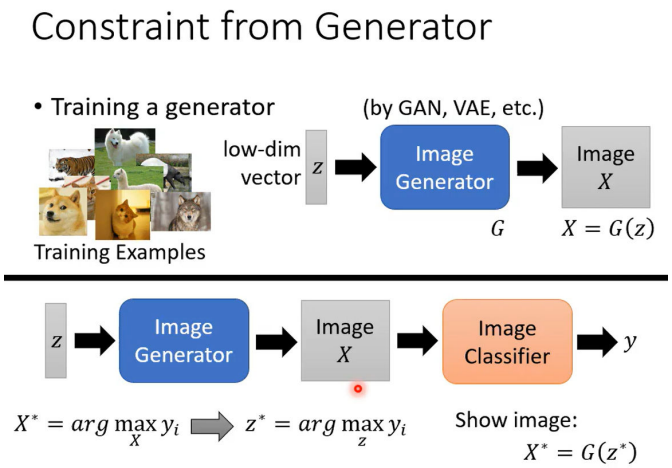
To answer how a network processes the input data, one way is visualization, i.e., we watch the output of hidden layers with the help of PCA, t-SNE, etc. (Attention seems to be good visual explanation, but there are opinions on both sides, c.f. https://arxiv.org/abs/1902.10186 and https://arxiv.org/abs/1908.04626.)

As watching by ourselves is limited, we may also use *probing*, i.e., we train a certain classifier with a certain hidden layer's output, and if it performs well, that output should have corresponding information. (E.g., we train a POS classifier with the embedding that one of BERT's block outputs, and if it achieves high accuracy, that embedding should have POS information.) We need to be careful with the classifier, as low accuracy can result from improper training as well. The probe is not necessarily a classifier. For example, to explain a speech recognition model, we can train a TTS model to reconstruct the input of the speech recognition model with one of its hidden layers' outputs, and if the TTS model can only reconstruct speech lack of certain information, we know there is no such information in that hidden layer's output, as demonstrated below.



Now let's turn to global explanation, which means explaining the whole model.

We can know what a filter in CNN detects in this way: notate that filter's outputs as a_{ij} and the input image as X, get $X^* = arg\ \max_X \sum_i \sum_j a_{ij}$ by gradient ascent, and the image X^* contains the patterns that filter detects. It seems possible to find what images of a certain class y_i look like by $X^* = arg\ \max_X y_i$, but it does not work as such an X^* is usually meaningless noises. (That is not surprising, considering what we will learn in *adversarial attack*.) To solve this problem, we may compute $X^* = arg\ \max_X \left(y_i + R(X)\right)$ where $R(X)$ measures how likely X is an expected image (e.g., $R(X) = -\sum_{i,j} |x_{ij}|$ measures how likely X is a black-and-white image of a digit as such an image should have only a few white pixels.). We can get better images with several regularization terms and hyperparameter tuning, c.f. https://arxiv.org/abs/1506.06579, which is quite difficult. A simplified way is as demonstrated below and performs well.



Of course, what we get with the methods above may not be what the machine really "think", as we try hard to find an explanation satisfying our expectation. Anyway, we just need to make our bosses, customers or ourselves comfortable!

In addition to what we have discussed, we can train an interpretable model (e.g., a linear one) to mimic the behavior of an uninterpretable model (or a part of it, c.f. *local interpretable model-agnostic explanation*, LIME).

Chapter 10 Adversarial Attack

Motivation

We seek to deploy neural networks in the real world. Therefore, the networks are required to be not only accurate, but also robust to those inputs built to fool them, which is useful for spam classification, malware detection, network intrusion detection, etc.

How to Attack

Take image classification as an example. The original image (a.k.a. a benign image) becomes an attacked one when it is changed a little (e.g., added with a small perturbation) to mislead the network. The non-targeted attack aims to make the network output anything other than the true class, while the targeted attack aims to make the network misclassify the image as a specific class.

Suppose we have an image x^0, a network f, and that $f(x^0) = y^0$. y^0 is most close to $\hat{y}$, a one-hot vector. To make a non-targeted attack, we choose $L(x) = -e(y, \hat{y})$, where $y = f(x)$. To make a targeted attack, we choose $L(x) = -e(y, \hat{y}) + e(y, y^{target})$. Then, we try to find $x^* = arg \min_{d(x^0, x) \leq \epsilon} L(x)$. ϵ here is to make sure the difference not perceived by humans, and we usually choose $d(x^0, x) = L_\infty(x - x^0)$ (L-infinity norm, not loss) instead of L-2 norm, as it is more perceivable to change one pixel much than to change some pixels a little.

To make an attack, we start from $x = x^0$ and update x (instead of parameters) with gradient descent. To let $d(x^0, x) \leq \epsilon$, we can fix an x out of bound by replacing it with the closest x under constraint.

An easier way is *Fast Gradient Sign Method* (FGSM, c.f. https://arxiv.org/abs/1412.6572), which updates x only once with the learning rate ϵ and gradient $g' = sgn(g)$. If we do that for more times, referred to as *Iterative FGSM* (c.f. https://arxiv.org/abs/1607.02533), it will behave better (Here x can go out of bound and need to be fixed.).

Black Box Attack

The attack approaches we have discussed rely on knowing the network f (to compute the gradient). Such an attack is referred to as a *white box attack*. Model parameters are not obtainable in most online API, but not releasing a model does not necessarily make it safe, as a *black box attack* is also possible.
If we have the training data of the target network, we may train a proxy network with it. Then we get attacked images of this proxy (as what we do for a white box attack), which can be used to attack the target network.
If we do not have the training data, we can get many input-output pairs of the target network, and use them to train a proxy.

Black box attacks are still easy to succeed (although more difficult than white box attack), especially for non-target attacks. The ensemble attack which uses more than one proxy performs better. (c.f. https://arxiv.org/pdf/1611.02770.pdf)
The reason why the attack is so easy is still unknown, but many researchers believe that an image will be wrongly classified when it (the vector representing it, to be strict) changes only a little to a certain direction, and such directions of different networks are similar, c.f. https://arxiv.org/pdf/1611.02770.pdf. Some researchers believe that such phenomenon result from the training data, and are not bugs but features, c.f. https://arxiv.org/abs/1905.02175.

There are some special attack approaches. The one pixel attack changes only one pixel to make an attacked image, c.f. https://arxiv.org/abs/1710.08864. The universal attack tries to attack all images by adding one same noise (attacked signal), c.f. https://arxiv.org/abs/1610.08401.

Not only image classification models can be attacked, but also speech processing models (e.g., we can mislead a synthesized-speech detector) and NLP models (c.f. https://arxiv.org/abs/1908.07125).

Attack in the Physical World

The attack we have discussed is in the digital world, and the attack in the physical world has more problems to consider. An attacker needs to find perturbations generalized beyond one single image. Moreover, extreme differences between adjacent pixels in the perturbation are unlikely to be captured accurately (because of the limitation of cameras), and perturbations should be comprised mostly of those colors reproducible (for the printer).
There have been many examples of attacks in the physical world.
Adversarial reprogramming (c.f. https://arxiv.org/abs/1806.11146) tries to make the model do a certain task at the attacker's will (instead of its original task).
The attack can even happen at the training phase, by adding something special (but still correctly labelled) into the training data. That will cause "backdoor" in the model, c.f. https://arxiv.org/abs/1804.00792. (Be careful with your database!)

Defense

There are two main way of defense, the passive one and the proactive one.
The passive defense adds a filter before the usual network to make the attack signal less harmful. The filter usually smoothens the input, but has a side effect of lower accuracy. There are other methods, such as image compression (c.f. https://arxiv.org/abs/1704.01155, https://arxiv.org/abs/1802.06816) or regenerate the input with a generator (c.f. https://arxiv.org/abs/1805.06605).
However, such a method is similar to adding a smoothening layer to the network, and can still be attacked easily once the attacker knows that. Therefore, randomization (e.g., introducing random resizing layers, random padding layers, or randomly selecting one pattern) is a better method, although it can still be broken through.
The Proactive defense is usually done by adversarial training. Notate the training set as $X = \{(x^1, \hat{y}^1), (x^2, \hat{y}^2), \ldots, (x^N, \hat{y}^N)\}$, and we train a model with X. Then we find the adversarial input (i.e., attacked image) $\tilde{x}^n$ for each x^n. Label them with their original labels, and we get a new training set $X' = \{(\tilde{x}_1, \hat{y}_1), (\tilde{x}_2, \hat{y}_2), \ldots, (\tilde{x}_N, \hat{y}_N)\}$. Then we use both X and X' to update the model. In fact, this method tries to find the problems of the model and fix them. It can be seen as data augmentation, and can also be used to avoid overfitting.
Adversarial training may lose effectiveness once the attack algorithm changes. It also needs a lot of computation, but there are methods to avoid that (c.f. Adversarial Training for Free, https://arxiv.org/abs/1904.12843)
Attack and defense methods are still evolving,

Chapter 11 Domain Adaptation

We have supposed that training and testing data are independent and identically distributed (i.i.d.), but it is not always the case. Training and testing data can have different distributions, which is referred to as *domain shift*. *Domain adaptation* tries to improve the model's performance when domain shift happens.
Domain adaptation can be seen as one part of *transfer learning*, which aims to enable the machine to do another task instead of the one it is trained for (c.f. https://youtu.be/qD6iD4TFsdQ).
There are different cases of domain shift. The probability distribution of labels can differ; similar features can have different labels; etc. We will only focus on the case where the inputs of training and testing data differ (e.g., training data is black-and-white images while testing data color ones). We refer to the domain of training data as the *source domain*, and that of testing data the *target domain*.

We will talk about different methods of domain adaptation for different knowledge of the target

domain we have.

With a lot of labeled data of the target domain, we can use it to train a model directly, and do not need domain adaptation.

With little labeled data of the target domain, we can use it to fine-tune the model (similar to what we did in Chapter 7). Be careful about overfitting as we do not have much data. We should not train too many iterations, and we may also choose smaller learning rates, put constraint on the change of the model after fine-tuning, etc.

A more common case is that we have a large amount of unlabeled data of the target domain. The basic idea is to train a feature extractor (a network) which learns to ignore the difference of the source and target domain, and get features of the same distribution from both domains.
An image classifier can be decomposed into a feature extractor (first a few layers) and a label predictor (the following layers, the number of the layers considered as a feature extractor is a hyperparameter), and we train a feature extractor with *domain adversarial training*. To be specific, we input to the feature extractor data from the source domain as well as the target domain, and get the output features. Then we train a domain classifier which tries to decide these features are from the source or target domain, while the feature extractor tries to fool it. (The extractor cannot output a same feature for all inputs, or it will not support the label predictor.) We may find its idea similar to that of GAN. That's to say, notate the loss of the label predictor (usually a cross-entropy) as L and the loss of the domain classifier as L_d, then the loss of the feature extractor is $L - L_d$. (It is quite difficult to train successfully!)
However, there is still a shortcoming that the domain classifier may try to classify images of the source domain as from the target domain and vice versa (to achieve high L_d). Moreover, we should consider the decision boundaries (learned from the source domain) and try to keep the target data as far as possible away from them. One possible way is to keep the entropy of the label predictor's output small (i.e., avoid the probability distribution vector having similar values at multiple classes). There are better and more complicated methods, such as DIRT-T (c.f. https://arxivporg/abs/1802.08735) and Maximum Classifier Discrepancy (c.f. https://arxivporg/abs/1712.02560)

The labels source and target domain have may not be the same, where universal domain adaptation (c.f. https://openaccess.thecvf.com/content_CVPR_2019/html/You_Universal_Domain_Adaptation_CVPR_2019_paper.html) comes into play.

With little unlabeled data of the target domain, we may use Testing Time Training (TTT, c.f. https://arxiv.org/abs/1909.13231).

Domain adaptation with no knowledge of the target domain is referred to as domain generalization. In some cases, the training set includes many different domains (but not the target domain), c.f. https://ieeexplore.ieee.org/document/8578664. In other cases, the training set includes one single domain. The idea to solve that is similar to that of data augmentation, c.f.

https://arxiv.org/abs/2003.13216.

Chapter 12 Deep Reinforcement Learning (RL)

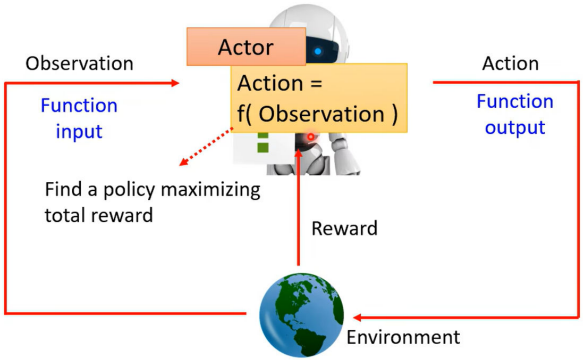
Deep Reinforcement Learning (RL) is a complicated technique with a wide range of applications (e.g., the famous AlphaGo). We will try to put it simple in this chapter.

What we have discussed is mostly *supervised learning* where we provide the machine with inputs as well as their expected outputs. (Self-supervised learning and auto encoder actually need labels automatically produced.) However, it is challenging to label data in some tasks while the machine can still know whether the result is good or not, where we need RL.

What is RL: Three Steps of ML

Markov Decision Process is usually introduced first when we talk about RL. However, we will try a different way, explaining RL based on the same three steps as usual machine learning (to define a function with unknowns, to define a loss function, optimization).

In RL, an *actor* makes an *observation* of the *environment*, takes an *action* based on it, and gets a *reward* from the environment. These three steps are repeated again and again. Therefore, the actor can be considered as a function which takes an observation as input and outputs an action, and RL is, roughly speaking, to look for such a function that maximizes the reward, as demonstrated below.



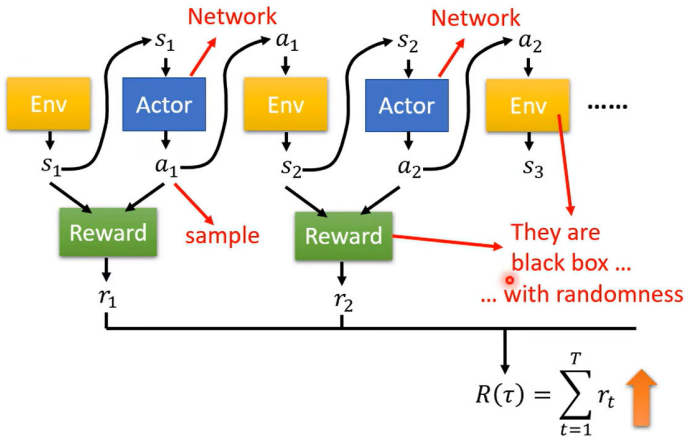
In fact, that process is similar to the three steps of usual ML.

Step 1, to define a function with unknowns. The actor is also referred to as the *policy* network (The actor in earlier days can be something else, e.g., a look-up table.), whose architecture can be CNN, RNN, transformer, etc., based on our need. The input of the network is the observation, represented as a vector (or matrix). In the output layer, each possible action corresponds to a neuron that outputs the score of the action. All these scores add up to 1, just as a classifier. (We may use Softmax to implement it,) We then sample an action to take, using the scores as probability (instead of taking the action of highest score, for randomness).

Step 2, to define a loss function. The actor starts with an observation s_1, takes an action a_1, gets

a reward r_1, and s_2, a_2, r_2, etc., until the game ends. The period of one single game is called an *episode*, and at the end of an episode, the actor gets a total reward (a.k.a. return) $R = \sum_{t=1}^{N} r_t$, which we want to maximize. Therefore, we can consider $-R$ as the loss function of RL.

Step 3, optimization. Notate the interaction between the actor and the environment as $\tau = \{s_1, a_1, s_2, a_2, \ldots\}$, where s_i is the observation from the environment and a_i is the action of the actor, we refer to τ as a trajectory. The reward r_i is the output of a function, which takes s_i and a_i as input, and $R = R(\tau)$ is to be maximized. It seems easy to solve, but although the actor is a network, the actions are sampled and the environment and the reward function are black boxes (even with randomness in some cases), as demonstrated below. Therefore, we cannot train the actor by gradient descent, and how to do the optimization is the main challenge in RL. (Although normal ML methods also have randomness in training, e.g., random initial, RL shows randomness in testing as well.)



Policy Gradient

Notate the output of the actor as $a = \theta(s)$, where θ is the parameter vector and s is the environment. To make the actor take (resp. not take) a specific action $\hat{a}$ (represented with a one-hot vector), we choose a loss function $L = e(a, \hat{a})$ (resp. $L = -e(a, \hat{a})$). Combining these two losses, we define the loss function as $L = \sum_{i=1}^{N} A_i e_i$, and $\theta^* = arg\min_{\theta} L$. We thus control the actor. It is actually supervised learning, but it is not easy to get such s-$\hat{a}$ pairs and decide As.

We will see a few different versions of actors.

Version 0. We initialize an actor, let it interact with the environment, and use r_t as A_t for $(s_t, \hat{a}_t)$ ($\hat{a}_t$ is the action the initial actor takes when seeing s_t). However, such a method is short-sighted. In fact, an action will affect the subsequent observations and thus subsequent rewards. The actor has to sacrifice immediate reward to gain more long-term reward, referred to as *reward delay*. Therefore, the actor will not perform well with such a method.

Version 1. We define $G_t = \sum_{i=t}^{N} r_i$ (referred to as *cumulated reward*), and use G_t as A_t for $(s_t, \hat{a}_t)$. This method is better than version 0, but those rewards far later than the action may not be the credit of it.

Version 2. We introduce a discount factor γ ($\gamma < 1$), define $G_t^{'} = \sum_{i=t}^{N} \gamma^{i-t} r_i$ (referred to as *discounted cumulated reward*), and use $G_t^{'}$ as A_t for $(s_t, \hat{a}_t)$. (Earlier actions will consider more rewards with this method, but that is reasonable as earlier actions should be more important.)
Version 3. We can see that good or bad reward is relative (E.g., if all rewards are larger than 10, a reward of 10 is actually bad.). Therefore, we should minus A_t with a *baseline* b, so that As have positive as well as negative values. We will discuss how to decide b later.
The algorithm of *policy gradient* is shown below. We can see that data are collected in the training loop. In fact, once we update the parameters, the actor will take different actions, get different rewards and see different observations. Therefore, we have to obtain data again with the updated actor, although that will take a lot of time.

- Initialize actor network parameters θ^0
- For training iteration $i = 1$ to T
  - Using actor θ^{i-1} to interact
  - Obtain data $\{s_1, a_1\}, \{s_2, a_2\}, \ldots, \{s_N, a_N\}$
  - Compute $A_1, A_2, \ldots, A_N$
  - Compute loss L
  - $\theta^i \leftarrow \theta^{i-1} - \eta \nabla L$

Data collection is in the "for loop" of training iterations.

If the actor to train and the actor to interact are the same one (as what we have talked about), we refer to such learning as *on-policy learning*. Relatively, these two actors can be different, referred to as *off-policy learning*. Roughly speaking, the actor to train has to "know" its difference from the actor to interact in off-policy learning. One common as well as powerful method for off-policy learning is *Proximal Policy Optimization* (PPO), c.f. https://youtu.be/OAKAZhFmYoI.

Exploration is significant when collecting training data. The actor needs randomness, or we may not train successfully, as we can see the reward of an action only when it is taken. Therefore, we may enlarge the entropy of the actor's output, or add noise to the actor's parameters.

Actor-Critic

Given an actor θ, the *critic* will tell how good it is when observing s (and taking action a). The value function $V^{\theta}(s)$ is expected to predict the discounted cumulated reward after the actor θ observing s (without playing a whole episode), which can be used as the critic. We can see that the output of $V^{\theta}(s)$ is a scalar, decided by s as well as θ. There are two common approaches to estimate $V^{\theta}(s)$.
In the *Monte-Carlo* (MC) based approach, the critic watches the actor θ to interact with the environment for many episodes, and obtain many ss and their discounted cumulated reward $G^{'}$s (until the end of the episode). Then, we let $V^{\theta}(s_t)$ be as close to $G_t^{'}$ as possible.
In the *Temporal-difference* (TD) approach, we train $V^{\theta}(s)$ with many groups of (s_t, a_t, r_t, s_{t+1}), and let $V^{\theta}(s_t) - \gamma V^{\theta}(s_{t+1})$ be as close to r_t as possible. That is because $G_t^{'} = r_t + \gamma G_{t+1}^{'}$.
Compared with the MC approach, TD approach is less intuitive but takes less time (as the actor

does not need to play a whole episode).

These two approaches can have different results because the hypotheses behind them are different. (In the example below, MC assumes that s_a will lead to the s_b whose reward is 0, while TD supposes that s_b has no relation to s_a.)

- The critic has observed the following 8 episodes
  - $s_a, r = 0, s_b, r = 0$, END
  - $s_b, r = 1$, END
  - $s_b, r = 1$, END
  - $s_b, r = 1$, END
  - $s_b, r = 1$, END
  - $s_b, r = 1$, END
  - $s_b, r = 1$, END
  - $s_b, r = 0$, END

(Assume $\gamma = 1$, and the actions are ignored here.)

$V^\theta(s_b) = 3/4$

$V^\theta(s_a) =?$ 0? 3/4?

Monte-Carlo: $V^\theta(s_a) = 0$

Temporal-difference:

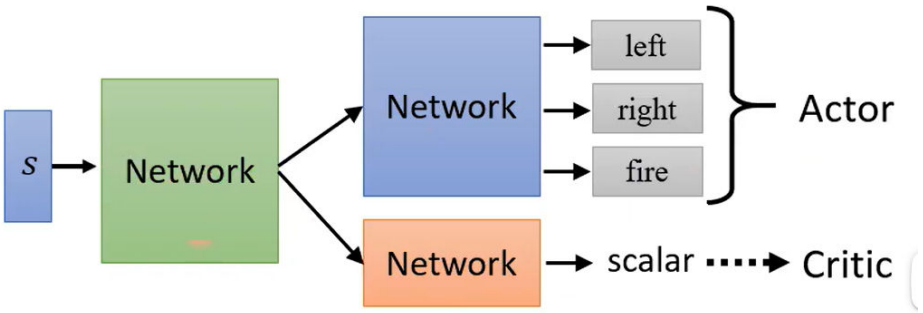
$V^\theta(s_a) = V^\theta(s_b) + r$

3/4 3/4 0

Now we have learned how to estimate $V^\theta(s)$, and we can use $b_t = V^\theta(s_t)$ as the baseline mentioned in policy gradient version 3, as $A_t = G_t^{'} - V^\theta(s_t)$ evaluates how good (s_t, a_t) is. In fact, as the action to take when observing s_t is sampled while $V^\theta(s_t)$ is decided by the cumulated discounted rewards of all possible actions, A_t will be positive if a_t (the action the actor takes) is better than average, and vice versa.

However, such a version still has a problem that $G_t^{'}$ is just the discounted cumulated reward of one certain sample but the rewards later ($r_{t+1}, r_{t+2}, \dots$) can differ. Therefore, we introduce version 4. We use $V^\theta(s_{t+1})$ instead to measure the rewards later, i.e., let $A_t = r_t + V^\theta(s_{t+1}) - V^\theta(s_t)$. This method is widely used, referred to as *Advantage Actor-Critic*.

The actor and the critic have the same input (s), so they can share some parameters, as demonstrated below.



It is possible to do RL with just the critic, e.g., *Deep Q Network* (DQN, c.f. https://youtu.be/o_g9JUMw1Oc, https://youtu.be/2-zGCx4iv_k, https://arxiv.org/abs/1710.02298).

Reward Shaping

If $r_t = 0$ in most cases (referred to as *sparse reward*, e.g., robot arm to bolt on the screws), we will not know whether actions are good or bad. We, as developers, need to define extra rewards to guide agents, referred to as *reward shaping*.

There have been many examples of reward shaping, c.f. https://openreview.net/forum?id=Hk3mPK5gg&noteId=Hk3mPK5gg,

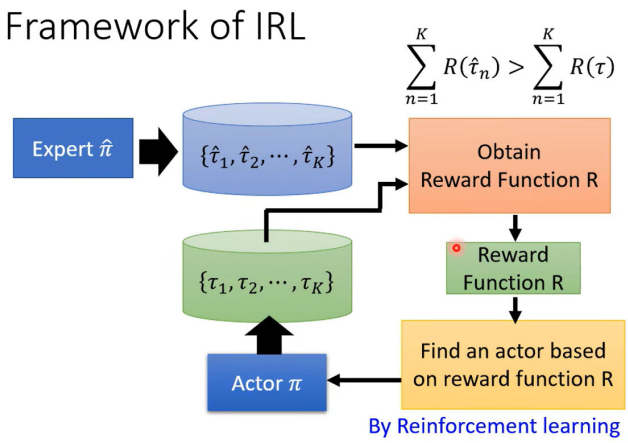
https://bair.berkeley.edu/blog/2017/12/20/reverse-curriculum/. We should be careful about reward shaping, as it needs quite domain knowledge.
Curiosity-based reward shaping (c.f. https://arxiv.org/abs/1705.05363) is an interesting and famous method. It means that the agent will obtain extra rewards when seeing something new (but meaningful, in case that the agent considers changing noises as "something new").

No reward: Learning from Demonstration

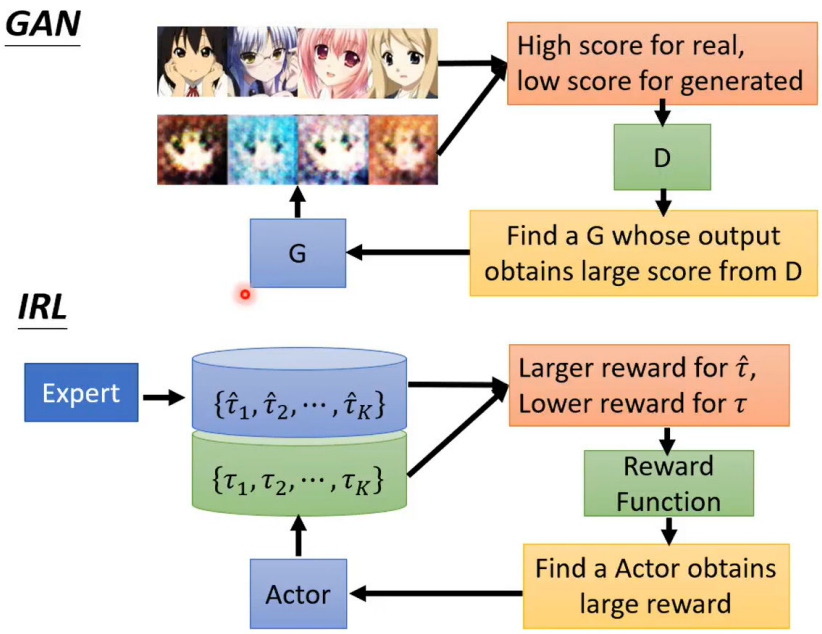
Even to define rewards can be challenging in some tasks, and hand-crafted rewards can lead to uncontrolled behavior. Therefore, we want to do RL with no reward.

One method is *imitation learning*. The actor can interact with the environment, but the reward function is not available. Instead, it learns by demonstration of the expert (usually humans) $\{\hat{\tau}_1, \hat{\tau}_2, \ldots, \hat{\tau}_k\}$, each $\hat{\tau}$ a trajectory of the expert. In fact, such a method can be considered as supervised learning, and is also referred to as *behavior cloning*. However, the experts only sample limited observations. Moreover, some irrelevant behaviors of the experts will also be copied, while some important behavior may not as the ability of the agent is limited.

Another method is *inverse reinforcement learning* (IRL), where the machine learns a reward function itself with demonstration of the expert. (Then we can use this reward function to find the optimal actor, as we do in usual RL.) Although such a reward function may not be complex, it does not mean that the agent will be too simple.
IRL follows the principle that the teacher is always the best. To be specific, we first initialize an actor; in each iteration, the actor interacts with the environment to obtain some trajectories, defines a reward function making the trajectories of the teacher (expert) better than those of the actor, and updates to maximize the reward based on the new reward function; finally, the (latest) reward function and the actor learned with it is output. The framework of IRL is as demonstrated below.



Considering the actor as a generator and the expert a discriminator, we can see that IRL is thus no different from GAN, as shown below.



What the actor learned by IRL will not be the same as the solution of the expert. In fact, maybe we can tune the final reward function (by hand) to make the actor better than the expert.

There is another one method referred to as *reinforcement learning with imagined goals* (RIG), in which the machine generates some goals by itself to achieve the user-specified goal, c.f. https://arxiv.org/abs/1807.04742 and https://arxiv.org/abs/1903.03698.

To Learn More – Deep Reinforcement Learning (Scratching the Surface)

This section is not included in ML2021, NTU, but also taught by Prof. Lee. Only key points are listed here.

1. Reinforcement learning is a huge topic, and as David Silver puts it, AI=RL+DL (RL+DL is also referred to as *deep reinforcement learning*).
2. RL can be used to learn a chat-bot. We may generate many dialogues with two agents, and use some pre-defined rules to evaluate these dialogues, c.f. https://arxiv.org/pdf/1606.01541v3.pdf.
3. RL may help in text generation, c.f. Hongyu Guo, Generating Text with Deep Reinforcement Learning, NIPS, 2015; Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, Wojciech Zaremba, Sequence Level Training with Recurrent Neural Networks, ICLR, 2016.
4. We can do RL by learning an actor, referred to as policy-based, or by learning a critic, referred to as value-based. These two can be used together, and *asynchronous advantage actor-critic* (A3C) is the best method now. (The performance of the famous *Deep Q Network* is worse.)

   AlphaGo uses policy-based, value-based, as well as model-based (Monte Carlo tree search) methods.

5. Resources for learning RL:
   The textbook Reinforcement Learning: An Introduction, https://webdocs.cs.ualberta.ca/~sutton/book/the-book.html
   Lectures of David Silver, http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching.html (10 lectures, 1:30 each); http://videolectures.net/rldm2015_silver_reinforcement_learning/ (Deep Reinforcement Learning)
   Lectures of John Schulman, https://youtu.be/aUrX-rP_ss4
6. The reward is not relevant to the actor's parameter θ, so it does not have to be differentiable.
7. To train an actor with simply total reward, we can do as follows:
   Consider a trajectory $\tau = \{s_1, a_1, r_1, s_2, a_2, r_2, \ldots, s_T, a_T, r_T\}$, we define total reward $R(\tau) = \sum_{i=1}^{T} r_i$. When we use an actor, each τ has a possibility to be sampled and this possibility depends on the actor's parameter θ. Therefore, the expectation of total reward $\overline{R}_\theta = \sum_{all\ possible\ \tau} R(\tau)P(\tau|\theta)$. In practice, we use the actor to obtain N trajectories and $\overline{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N} R(\tau^n)$.

   Then, we calculate the gradient of $\overline{R}_{\theta^i}$. We know that $\frac{dlog(f(x))}{dx} = \frac{1}{f(x)}\frac{df(x)}{dx}$, so

$$\nabla\overline{R}_\theta = \sum_\tau R(\tau)\nabla P(\tau|\theta) = \sum_\tau R(\tau)P(\tau|\theta)\frac{\nabla P(\tau|\theta)}{P(\tau|\theta)} = \sum_\tau R(\tau)P(\tau|\theta)\nabla logP(\tau|\theta)$$

   We know that

$$P(\tau|\theta) = p(s_1)\prod_{t=1}^{T} p(a_t|s_t,\theta)p(r_t, s_{t+1}|s_t, a_t)$$

   so

$$logP(\tau|\theta) = logp(s_1) + \sum_{t=1}^{T}\left(logp(a_t|s_t,\theta) + logp(r_t, s_{t+1}|s_t, a_t)\right)$$

   Ignore the terms not related to θ, and

$$\nabla logP(\tau|\theta) = \sum_{t=1}^{T}\nabla logp(a_t|s_t,\theta)$$

   Therefore,

$$\nabla\overline{R}_\theta = \sum_\tau R(\tau)P(\tau|\theta)\nabla logP(\tau|\theta)$$

$$\approx \frac{1}{N}\sum_{n=1}^{N} R(\tau^n)\nabla logp(\tau^n|\theta)$$

$$= \frac{1}{N}\sum_{n=1}^{N} R(\tau^n)\sum_{t=1}^{T_n} log\nabla p(a_t^n|s_t^n,\theta)$$

$$= \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} R(\tau^n) log \nabla p(a_t^n | s_t^n, \theta)$$

Intuitively, this result considers the cumulative reward of the whole trajectory, which is very important. As $log \nabla p(a_t^n | s_t^n, \theta) = \frac{\nabla p(a_t^n | s_t^n, \theta)}{p(a_t^n | s_t^n, \theta)}$, we in fact do normalization so that the actor will not tend to take those more frequent actions (because of higher reward).

Now we can update θ by gradient ascent: $\theta^{i+1} \leftarrow \theta^i + \eta \nabla \overline{R}_{\theta^i}$.

If all rewards are positive, the actor will tend to take all the actions it has sampled as they all have positive rewards, and those actions not sampled will thus be taken less even though they are actually better. Therefore, it is better to add a baseline.

8. It is possible to learn an actor with simply critic, e.g., Q-Learning.

Chapter 13 Life-Long Learning

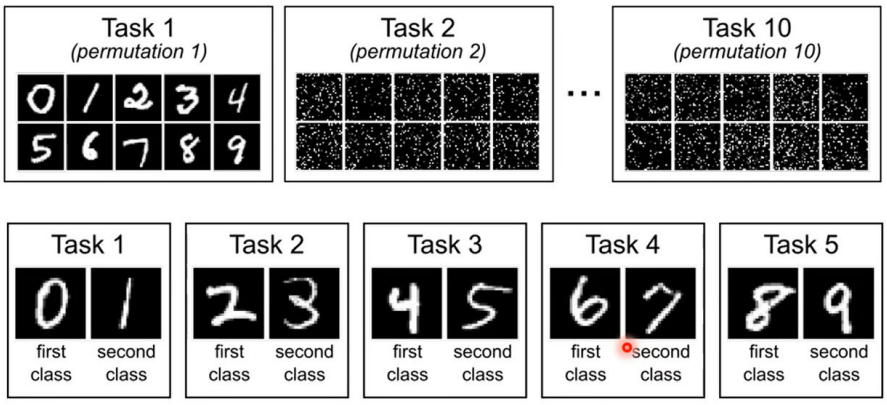
Not only humans but also AI needs *life-long learning* (LLL, a.k.a. continuous learning, never ending learning, incremental learning). We expect the model to learn more and more tasks.
In real-world applications, the model is first trained by labeled data, and then updated (from the parameters now, instead of random initialized ones) according to users' feedback continually. Here the first training can be considered as the original task and later training the new tasks.
Exactly speaking, the "different tasks" researchers talk about nowadays are usually one same task in different domains.

Problem

However, it is not easy to realize life-long learning. Suppose that we have two tasks, and we train a model with data of task 1. Then we continue to train this model with data of task 2, and we will see that the accuracy on task 2 increases while that on task 1 decreases. You may suspect that this model cannot do both tasks, but if we mix data of both task 1 and task 2 to train the same model (from scratch), the model may achieve both high accuracy on two tasks.
Such phenomenon is not rare. We can see that the machine is able do it but just don't do it. We refer to such phenomenon as catastrophic forgetting.
Although multi-task training (to train with mixed data) can solve the problem, it is expensive in terms of computation and storage. Instead, multi-task training is usually considered as the upper bound of LLL's performance. We do not train one model for each task, or we will eventually run out of storage, and that knowledge cannot transfer across different tasks.
Both transfer learning and life-long learning involve two (or more) tasks, but the former one aims to do task 2 based on task 1 and does not care whether the machine can still do task 1, while the latter one aims to not forget task 1 even though the machine has learned task 2.

Evaluation

To evaluate LLL, we need a sequence of tasks (one example is as demonstrated below). Then we calculate $R_{i,j}(i = 0,1,2, \ldots, T, j = 1,2,3, \ldots, T, T \text{ is the number of tasks})$ representing the performance on task j after training task i ($i = 0$ represents randomly initialized model without training), as demonstrated below the first picture. When $i > j$, $R_{i,j}$ shows whether task j is forgotten after training task i. When $i < j$, $R_{i,j}$ shows whether we can transfer the knowledge of task i to task j. To evaluate LLL, we calculate $\frac{1}{T}\sum_{i=1}^{T} R_{T,i}$ (referred to as accuracy) or $\frac{1}{T-1}\sum_{i=1}^{T-1}\left(R_{T,i} - R_{i,i}\right)$ (referred to as backward transfer, usually negative). (We calculate "forward accuracy" $\frac{1}{T-1}\sum_{i=2}^{T}\left(R_{i-1,i} - R_{0,i}\right)$ to evaluate transfer learning.)



		Test on			
		Task 1	Task 2	.....	Task T
Rand Init.		$R_{0,1}$	$R_{0,2}$		$R_{0,T}$
After Training	Task 1	$R_{1,1}$	$R_{1,2}$		$R_{1,T}$
	Task 2	$R_{2,1}$	$R_{2,2}$		$R_{2,T}$
	⋮				
	Task T-1	$R_{T-1,1}$	$R_{T-1,2}$		$R_{T-1,T}$
	Task T	$R_{T,1}$	$R_{T,2}$		$R_{T,T}$

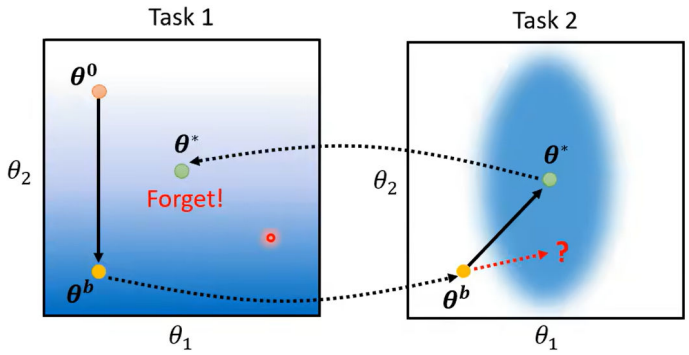
Research Direction

Selective Synaptic Plasticity

The first research direction of LLL we are going to talk about, as well as the most mature one, is

selective synaptic plasticity, a.k.a. *regularization-based approach*.

Catastrophic forgetting results from different tasks' error surface having different minimum points, as demonstrated below.



The error surfaces of tasks 1 & 2.
(darker = smaller loss)

The basic idea of selective synaptic plasticity is that some of the parameters are important to the previous tasks and we only change the unimportant ones to learn a new task. Notate the previous model as $\boldsymbol{\theta}^b$, we set a "guard" b_i for each parameter $\boldsymbol{\theta}_i^b$, and use $L^{'}(\boldsymbol{\theta}) = L(\boldsymbol{\theta}) + \lambda \sum_i b_i \left(\theta_i - \theta_i^b\right)^2$ as loss function of the new task, instead of usual $L(\boldsymbol{\theta})$. The value of b_i represents how much we want the corresponding parameter to be close to its previous value, and $\boldsymbol{\theta}$ is thus close to $\boldsymbol{\theta}^b$ in certain directions.

Here b_is are hyperparameters. A too small b_i may lead to catastrophic forgetting, while a too large one may lead to *intransigence*. To decide the value of b_i, roughly speaking, we watch how much the previous loss changes when θ_i^b change (If loss changes significantly, b_i should be large, and the model thus tends to avoid change this parameter, and vice versa.). There are many methods to decide the value of b_i, e.g., elastic weight consolidation (EWC, c.f. https://arxiv.org/abs/1612.00796), synaptic intelligence (SI, c.f. https://arxiv.org/abs/1703.04200), memory aware synapses (MAS, c.f. https://arxiv.org/abs/1711.09601), RWalk (c.f. https://arxiv.org/abs/1801.10112), sliced Cramer preservation (SCP, c.f. https://openreview.net/forum?id=BJge3TNKwH).

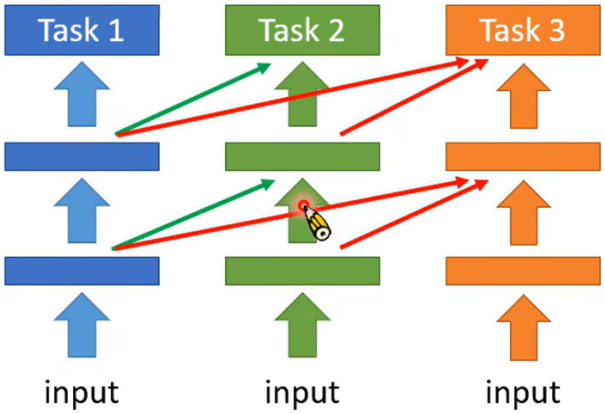
Gradient episodic memory (GEM, c.f. https://arxiv.org/abs/1706.08840) is another method for LLL, which is old but effective. Notate the gradient of the current (new) task as $\boldsymbol{g}$ and that of previous tasks as $\boldsymbol{g}_b$, and we do gradient descent with $\boldsymbol{g}^{'}$, which is as close to $\boldsymbol{g}$ as possible while satisfying $\boldsymbol{g}_b \cdot \boldsymbol{g}^{'} \geq 0$. We may see that this method needs a little data of previous tasks, which is not what LLL exactly wants. (It is acceptable. In fact, regularization-based approaches also need extra memory.)

Additional resource allocation

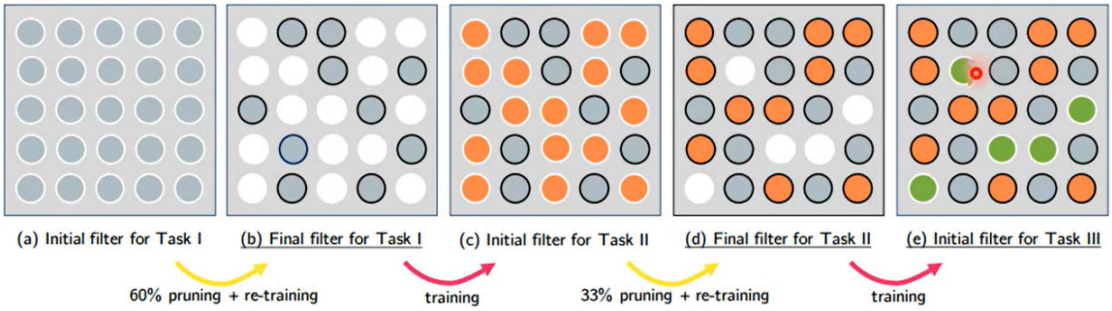
The second research direction of LLL is *additional neural resource allocation*.

The earliest idea is *progressive neural networks* (c.f. https://arxiv.org/abs/1606.04671). It creates new layers for the new task, and new hidden layers take their corresponding layers' output as input, as demonstrated below. It works well with a few tasks, but we will run out of storage with

too many tasks.



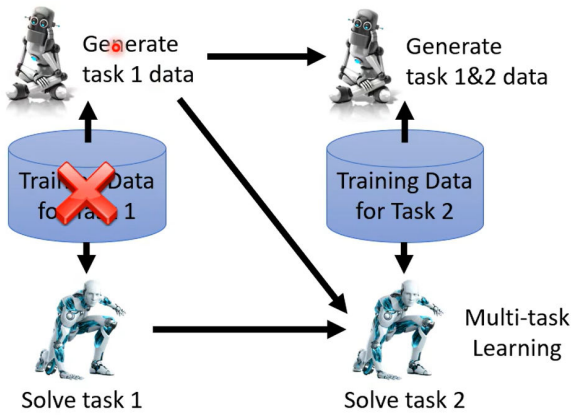
The idea of *PackNet* (c.f. https://arxiv.org/abs/1711.05769) is opposite. It does pruning and re-training for each task, and the number of parameters later tasks can use is smaller, as demonstrated below.



Compacting, picking and growing (CPG, c.f. https://arxiv.abs/org/1910.06562) is the combination of progressive neural networks and PackNet.

Memory Replay

The third research direction of LLL is *memory replay* (c.f. https://arxiv.org/abs/1705.08690, https://arxiv.org/abs/1711.10563, https://arxiv.org/abs/1909.03329). We train not only a model for the tasks, but also a generator generating pseudo data of all tasks. Then, we use these pseudo data (they are data of previous tasks now) as well as data of the current task to train the next model, as demonstrated below. With this method, we may get close to the upper bound of LLL.



Other Scenarios

We have been using the same network for different tasks. Obviously, it is not always the case.
For a classification model, we may expect it to learn tasks with different numbers of tasks, c.f. Learning without Forgetting (LwF), https://arxiv.org/abs/1606.09282; iCaRL: Incremental Classifier and Representation Learning, https://arxiv.org/abs/1611.07725.
In fact, what we have talked about is the simplest scenario of LLL. There are (maybe) 3 scenarios in total, c.f. https://arxiv.org/abs/1904.07734.

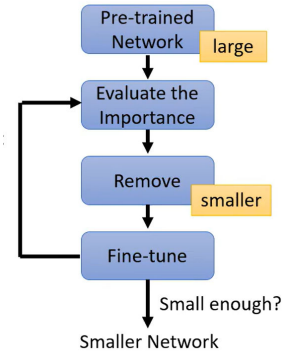
Moreover, catastrophic forgetting may not happen if we arrange certain tasks in a certain order. Such learning is referred to as *curriculum learning*.

Chapter 14 Network Compression

We have discussed some large models in earlier chapters. In this chapter, we are going to compress them, i.e., to reduce the number of parameters without worsening the model's performance. It is referred to as *network compression*, which helps deploy ML models in resource-constraint environments (We need to deploy ML models in edge devices for privacy, low latency, etc.).
We will talk about only software solution (no hardware solution) in this chapter.

Network Pruning

Networks are typically over-parameterized, i.e., there are significant redundant weights or neurons, and we can prune them just as what happens in human brains during childhood.
Roughly speaking, we do network pruning with following steps, as demonstrated below: 1st, pre-train a (large) network; 2nd, evaluate the importance of weights or neurons; 3rd, remove less important weights or neurons; 4th, fine-tune on training data; the 2nd to 4th steps can be repeated a few times until the network is small enough.



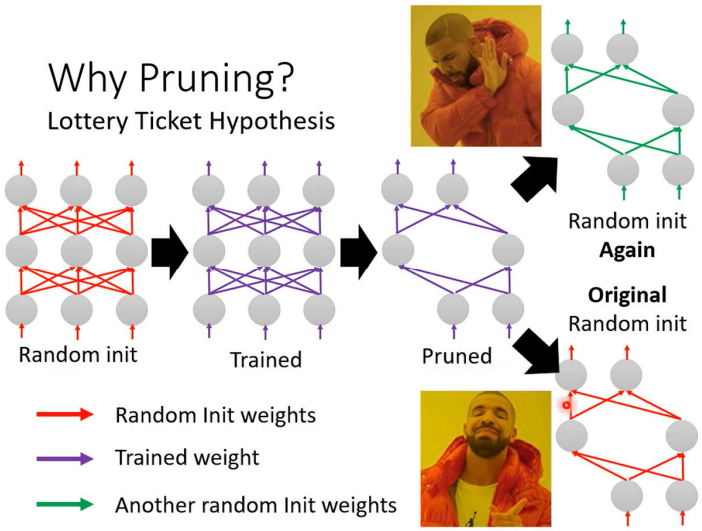
To evaluate the importance of a weight, we may watch its absolute value, apply life-long learning

(c.f. Chapter 14), etc.; to evaluate that of a neuron, we may watch the number of times it is not 0 on a given data set, etc. The accuracy of the network will drop (not too much, hopefully) after pruning, and we fine-tune it to recover. We cannot prune too much at a time, or the network will not recover.

Weight pruning may lead to practical issues, as the network architecture becomes irregular and is hard to implement and speedup (Some may set the weights pruned as 0 to facilitate GPU speedup, but that does not compress the network.). In fact, although we can often prone the majority of weights, the speed becomes lower in most cases. Oppositely, the network architecture after neuron pruning is still regular and thus easy to implement and speedup.

We cannot simply train a smaller network, as it is widely known that smaller networks are more difficult to learn successfully. Larger networks seem easier to optimize, and *lottery ticket hypothesis* (c.f. https://arxiv.org/abs/1803.03635) provides a possible explanation.

Lottery ticket hypothesis says that to train a large network is equivalent to train a number of small sub-networks and if one of them learns successfully, so will the large network. Then, as whether a network succeeds depends mostly on its initial weights, more sub-networks means larger success chance. In fact, if we train a large network and prune it, and then train the pruned network again, we will find that the training may success with the same initial weights as before while fail with weights randomly initialized again, as demonstrated below. That happens maybe because the initial parameters of the pruned network are “lucky”, i.e., these parameters enable the network to succeed.



A further study (*Deconstructing Lottery Tickets: Zeros, Signs and the Supermask*, c.f. https://arxiv.org/abs/1905.01067) finds that the large-final strategy the magnitude-increase strategy are two good strategies for pruning. It also says that the signs of initial weights are critic (“sign-ificance”, i.e., if a network can learn successfully with certain initial weights, it can still succeed when we change those initial weights’ values but keep their sign.). Moreover, it is possible that some of the sub-networks have initial weights so “lucky” that they do not need training at all, and we can just prune (weights) from a network randomly initialized. (Such networks are called weight agnostic neural networks in another research, c.f. https://arxiv.org/abs/1906.04358.)

However, there are also researches against lottery ticket hypothesis. Researchers find that it is

also possible to train the pruned network with weights randomly initialized again and achieve similar accuracy with some more epochs. They believe that lottery ticket hypothesis only applies to cases where the learning rate is small and the network is unstructured (because of weight pruning).

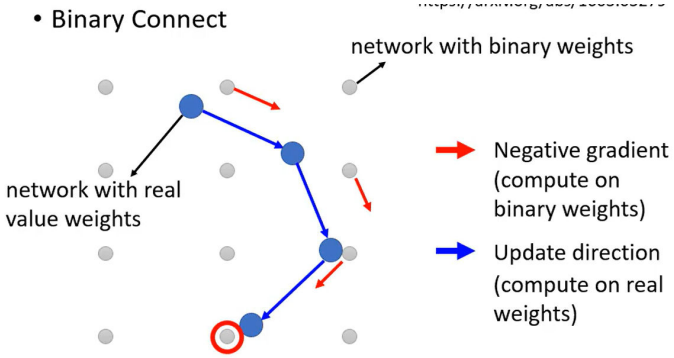
Knowledge Distillation

In *knowledge distillation* (c.f. https://arxiv.org/pdf/1503.02531.pdf and https://arxiv.org/pdf/1312.6184.pdf), we train a large network, referred to as *teacher net*, and get a small one, referred to as *student net*, based on it. We will take classification as an example.
The student net is trained with the corresponding output of the teacher network as ground truth. In other words, when training the student net, we minimize the cross-entropy between its output and the output of the teacher net (a vector representing probability distribution, instead of a one-hot vector) with the same input.
The student net usually performs better than the same network trained from scratch, as teacher net provides extra information (e.g., the relationship between two classes) The student net can even learn a class without any data belonging to it (We still need data from all classes to train the teacher net.).
It is a common practice that we average many models to get the result (by voting or averaging), referred to as *ensemble*. We can use multiple networks as the teacher net and thus do ensemble.
We may use $\sigma(x_i) = \frac{exp(x_i/T)}{\sum_j exp(x_j/T)}$ here, instead of usual Softmax, where T is a hyperparameter called "temperature". This new function can "smoothen" the probability distribution so that more information is reserved.
We may add some constraints (e.g., one certain layer of the student net should resemble another certain layer of the teacher net), which is usually helpful.
We may introduce a few more nets to link the teacher net and the student net.

Parameter Quantization

Network pruning and knowledge distillation aim to reduce the number of weights, and if we want to reduce the memory each weight occupies, we may try *parameter quantization*.
There are some different methods for parameter quantization.
We can use fewer bits to represents a value. That sacrifices the accuracy of a value but does not affect much. In fact, the network can sometimes perform better with less accurate values.
We can do *weight clustering*. That is to say, we divide the weights of similar values into several groups (the number of groups is chosen in advance), and replace the weights of the same group with their average.
We can also represent frequent weights with fewer bits and rare ones with more bits, e.g., *Huffman encoding*.
We can even train a network whose weights are binary (either +1 or -1), as demonstrated below.

Researches show that binary networks is less possible to overfit and performs similarly well compared with usual networks. (c.f. binary connect, https://arxiv.org/abs/1511.00363; binary network, https://arxiv.org/abs/1602.02830; XNOR-net, https://arxiv.org/abs/1603.05279.)



Architecture Design

We have learned standard CNN earlier, and we are going to talk about *depthwise separable CNN* as an example of *architecture design*.

In depthwise convolution, each filter considers one channel (instead of all channels in standard CNN). Therefore, the number of filters and that of output channels equal to the number of input channels, and the filters are $k \times k$ matrices. We can see that there is no interaction among different channels in depthwise convolution, so we need pointwise convolution, which is the same as standard CNN except that all filters are $1 \times 1 (\times \text{number of channels})$.

Notate the number of input channels as I, the number of output channels as O, and kernel size as $k \times k$, and a standard convolutional layer has $I \times k \times k \times O$ parameters while a depthwise convolutional layer and a pointwise one have $I \times k \times k + I \times O$ parameters. As O is usually a large number, we can see that

$$\frac{I \times k \times k \times O}{I \times k \times k + I \times O} = \frac{1}{O} + \frac{1}{k \times k} \approx \frac{1}{k \times k}$$

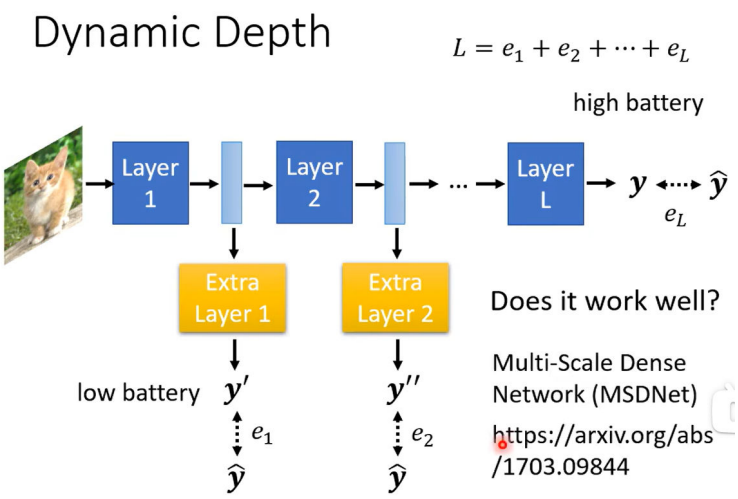
The idea of this method comes from linear networks. For a (fully-connected) linear layer with M inputs and N outputs, the weight $\boldsymbol{W}$ is a $N \times M$ matrix. We can divide it into two layers, the first one having K outputs and the second one K inputs. Then, the first layer's weight $\boldsymbol{U}$ is a $K \times M$ matrix and the second layer's weight $\boldsymbol{V}$ is a $N \times K$ matrix, and we thus reduce the number of weights. We can see that $\boldsymbol{W} = \boldsymbol{V}\boldsymbol{U}$, so $rank(\boldsymbol{W}) \leq K$, and such a method is referred to as *low rank approximation*.

There are many other researches on network architecture design, e.g., SqueezeNet, c.f. https://arxiv.org/abs/1602.07360; MobileNet, c.f. https://arxiv.org/abs/1704.04861; ShuffleNet, c.f. https://arxiv.org/abs/1707.01083; Xception, c.f. https://arxiv.org/abs/1610.02357; GhostNet, c.f. https://arxiv.org/abs/1911.11907.

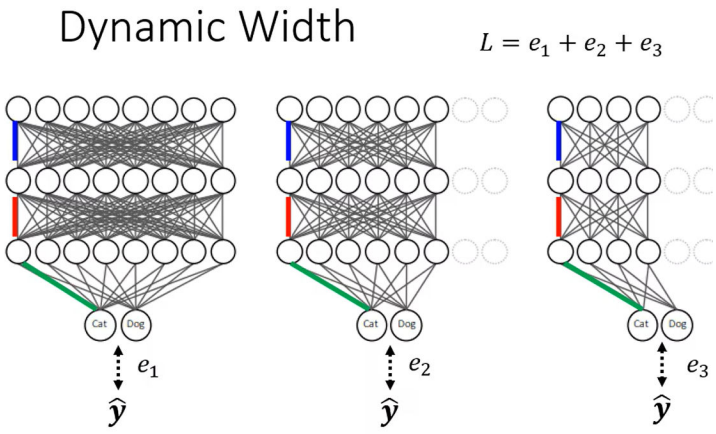
Dynamic Computation

Those four methods we have talked about aim to reduce the size of the network, while *dynamic computation* allows the network to adjust the computation it needs (according to the condition of the device). (We do not prepare a set of models for different condition, as it will cost too much memory.)

One method for dynamic computation is dynamic depth. Taking image classification as an example, we add extra layers that take the output of hidden layers as input and output a probability distribution vector (just as the final layer), and minimize the sum of all those vectors' cross-entropy with the ground truth, as demonstrated below. Multi-scale dense network works better, c.f. https://arxiv.org/abs/1703.09844.



Another method is dynamic width, where the network uses different numbers of weights, as demonstrated below. Slimmable neural network works better, c.f. https://arxiv.org/abs/1812.08928.



Moreover, we can let the network to decide the computation itself. For example, a classification network may decide the computation based on how difficult it is to classify a sample. C.f. SkipNet: Learning Dynamic Routing in Convolutional Networks; Runtime Neural Pruning; BlockDrop: Dynamic Inference Paths in Residual Networks.

Chapter 15 Meta Learning

Meta learning means learning to learn.

We all know that hyperparameters are significant for deep learning. In industry, people use a great number of GPUs to try a great number of sets of hyperparameters. In academia, however, we may have to "telepathize" a set of good hyperparameters. Therefore, we hope the machine to automatically determine the hyperparameters, which can be realized with machine learning.

Introduction

In terms of usual machine learning (or deep learning), we define a function f with unknown parameters $\boldsymbol{\theta}$, then define a loss function $L(\boldsymbol{\theta})$, and try to find the best parameters $\boldsymbol{\theta}^* = arg\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$ by gradient descent (or similar methods). We thus get a function $f_{\boldsymbol{\theta}^*}$, or f^*, learned by learning algorithm from data.

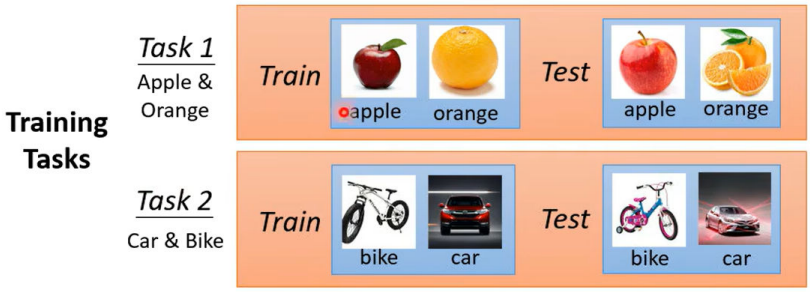
In fact, the learning algorithm F itself can be considered as a function (exactly, a functional), which takes the data (all training examples) as input and outputs the best function f^*. We can see that although f^* is learned, F is hand-crafted. Meta learning aims to learn F following the same three steps as usual machine learning (to define a function with unknowns, to define a loss function, optimization).

The first step is to find out what is learnable in a learning algorithm. The network architecture, initial parameters, learning rate, etc. are all learnable components of deep learning. Notate all these components we want the machine to learn as ϕ, and F_ϕ is the function with unknowns.

Certainly, the components to learn is not always the same. We usually categorize meta learning based on what is learnable.

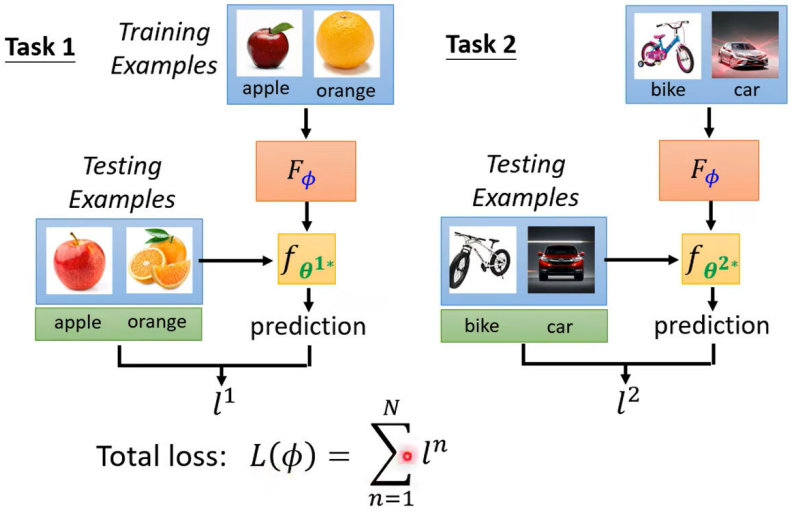
The second step is to define a loss function $L(\phi)$ for F_ϕ.

We use tasks to train F_ϕ, and we need both training data and testing data of each training task. (E.g., if we want to find a learning algorithm for binary classification, we need training data and testing data of several binary classification tasks, as demonstrated below.)



For any task i, we learn a function (a model) $f_{\boldsymbol{\theta}^{i*}}$ where $\boldsymbol{\theta}^{i*}$ is parameters learned by F_ϕ with the training data of task i. Then, we compute the difference between $f_{\boldsymbol{\theta}^{i*}}$'s output and the

ground truth with the testing data of task i, and notate it as l^i (for a classification task, l^i is the sum of cross-entropies). We define $L(\phi) = \sum_i l^i$, as demonstrated below.

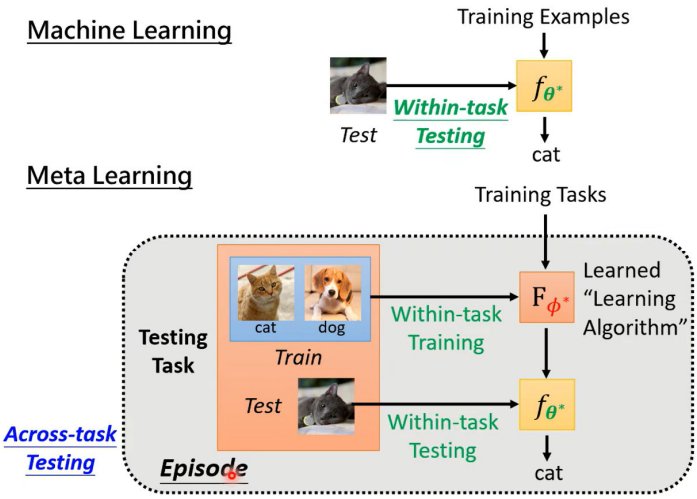


The third step is to find the ϕ that minimize $L(\phi)$, i.e., to find $\phi^* = arg\min_\phi L(\phi)$. If we are able to compute $\frac{\partial L(\phi)}{\partial \phi}$, we can simply use gradient descent. If $L(\phi)$ is not differentiable, we may use RL or other methods such as the evolutionary algorithm. Thus, we get F_{ϕ^*}, a learned learning algorithm.

Then we can apply F_{ϕ^*} to the testing task, which we care about. With F_{ϕ^*}, we need little training data to get f_{θ^*}. We thus realize few-shot learning.

ML v.s. Meta Learning

There are some differences between ML and meta learning.
Roughly speaking, ML aims to find a function f, while meta learning aims to find a function F that finds a function f.
There is usually one single task in ML, with its training data and testing data. Meta learning, however, involves training tasks and testing tasks, each having its own training data (a.k.a. support set) and testing data (a.k.a. query set). Therefore, the training (testing) of ML is also referred to as within-task training (resp. testing), while that of meta learning referred to as across-task training (resp. testing). The whole framework is as demonstrated below.



To compute $L(\phi)$, we need to compute the loss of each task, which needs both within-task training and within-task testing. Therefore, across-task training is also referred to as outer loop (often used in "learning to initialize"), while within-task training referred to as inner loop.

Meta learning and ML also have some resemblance.
What we have learned about ML can usually apply to meta learning, e.g., overfitting on training tasks, to get more training tasks to improve performance, task augmentation, etc.
There are also hyperparameters in learning a learning algorithm, but we only have to find one set of good hyperparameters for all similar tasks. Meta learning also needs development tasks when searching hyperparameters, just as the development set in ML. (Many meta learning researches ignore the necessity of development tasks. That may change in the future.)

What is Learnable in Meta Learning

Learn to Initialize

We know that f's initial parameters $\boldsymbol{\theta}_0$ has an effect on the performance. There are some methods to learn to initialize, of which *model-agnostic meta learning* (MAML) is the most famous one. It has a variation *reptile* (c.f. https://arxiv.org/abs/1803.02999). It is difficult to train MAML, c.f. Antreas Antoniou, Harrison Edwards, Amos Storkey, How to Train your MAML, ICLR, 2019.
We may recall that the pre-training in self-supervised learning can also find a good initialization. It trains the model first by proxy tasks (fill-in the blanks, etc.) with unlabeled data. In fact, an earlier and more typical way of pre-training is to mix data of different tasks to train a model, and use its parameters as a good initialization. Such a method is also referred to as multi-task learning, which is usually considered as the baseline of meta learning.
There are two possible reasons why MAML works. One is that the good initialization it finds can rapidly learn, and the other is that that initialization is close to the best parameters of tasks. One research (Rapid Learning or Feature Reuse? Towards Understanding the Effectiveness of MAML) shows that the latter one is the truth, and introduces an advanced method *almost no inner loop*

(ANIL).

More mathematical details behind MAML can be found in https://youtu.be/mxqzGwP_Qys; *first order MAML* (FOMAML) can be found in https://youtu.be/3z997JhL9Oo; reptile can be found in https://youtu.be/9jJe2AD35P8.

Optimizer

The optimizer (e.g., learning rate and momentum) is learnable, c.f. Marcin Andrychowicz, et al., Learning to Learn by Gradient Descent by Gradient Descent, NIPS, 2016.

Network Architecture Search (NAS)

Network structure is learnable, and to learn it is also referred to as *network architecture search* (NAS).

Obviously, $L(\phi)$ is not differentiable when ϕ represents a network's architecture (hyperparameters). One solution is RL, c.f. Barret Zoph, et al., Neural Architecture Search with Reinforcement Learning, ICLR, 2017; Barret Zoph, et al., Learning Transferable Architectures for Scalable Image Recognition, CVPR, 2018; Hieu Pham, et al., Efficient Neural Architecture Search via Parameter Sharing, ICML 2018. To be specific, we train an agent using a set of actions to determine the network architecture, i.e., output ϕ. Then, we train a network decided by this ϕ (within-task training), and define its accuracy as $-L(\phi)$, which is the reward to maximize in RL (cross-task training).

Another solution is Evolution Algorithm, c.f. Esteban Real, et al., Large-Scale Evolution of Image Classifiers, ICML 2017; Esteban Real, et al., Regularized Evolution for Image Classifier Architecture Search, AAAI, 2019; Hanxiao Liu, et al., Hierarchical Representations for Efficient Architecture Search, ICLR, 2018.

We may also define differentiable $L(\phi)$, c.f. Hanxiao Liu, et al., DARTS: Differentiable Architecture Search, ICLR, 2019.

Data Processing

Data augmentation is learnable, c.f. Yonggang Li, Guosheng Hu, Yongtao Wang, Timothy Hospedales, Neil M. Robertson, Yongxin Yang, DADA: Differentiable Automatic Data Augmentation, ECCV, 2020; Daniel Ho, Eric Liang, Ion Stoica, Pieter Abbeel, Xi Chen, Population Based Augmentation: Efficient Learning of Augmentation Policy Schedules, ICML, 2019; Ekin D. Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, Quoc V. Le, AutoAugment: Learning Augmentation Policies from Data, CVPR, 2019.

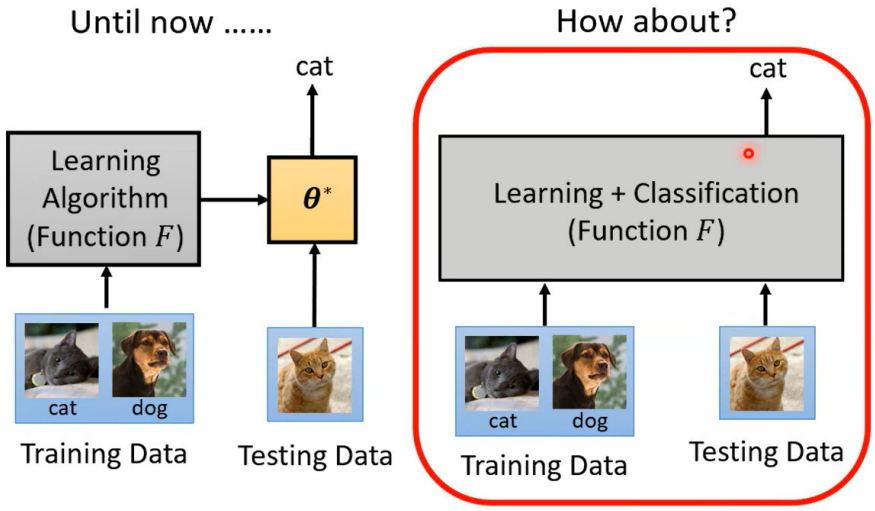
We sometimes need sample reweighting, i.e., give different samples different weights. That is not easy. Taking binary classification as an example, samples close to the boundary may be given larger weights to focus on tough examples, or smaller weights to ignore noisy labels. Sample weighting strategy is learnable, c.f. Jun Shu, Qi Xie, Lixuan Yi, Qian Zhao, Sanping Zhou, Zongben Xu, Deyu Meng, Meta-Weight-Net: Learning an Explicit Mapping for Sample Weighting, NeurIPS,

2019; Mengye Ren, Wenyuan Zeng, Bin Yang, Raquel Urtasun, Learning to Reweight Examples for Robust Deep Learning, ICML, 2018.

Beyond Gradient Descent

We can see that those learnable we have discussed are all based on gradient descent. If ϕ provides us with a network which can be optimized (i.e., get $\boldsymbol{\theta}^*$) using training data in a certain way (instead of gradient descent), we actually invent a new learning algorithm! Researchers have been trying to do that, c.f. Andrei A. Rusu, Dushyant Rao, Jakub Sygnowski, Oriol Vinyals, Razvan Pascanu, Simon Osindero, Raia Hadsell, Meta-Learning with Latent Embedding Optimization, ICLR, 2019.

We have always been using training data by a certain learning algorithm (F) to get $\boldsymbol{\theta}^*$, and applying it to testing data. However, in classification, we may put learning and classification together as F, and get the labels of testing data directly, as demonstrated below. Such a method is referred to as learning to compare, a.k.a. the metric-based approach, c.f. https://youtu.be/yyKaACh_j3M, https://youtu.be/scK2EIT7klw, https://youtu.be/semSxPP2Yzg, https://youtu.be/ePimv_k-H24.



Applications

One application of meta learning is few-shot classification, where we have only a few examples for each class (N-ways K-shot classification means N classes with K examples each.) To do that with meta learning, we need many N-ways K-shot tasks for training and testing, whose data can be sampled from Omniglot, a database having 1623 characters, 20 examples each.

There are more applications, c.f. http://speech.ee.ntu.edu.tw/~tlkagk/meta_learning_table.pdf.

Epilogue

What do we learn?

At the beginning, we learn that machine learning is not magic, but to find a function. In Chapter 1 and 2, we see linear models are not enough and learn about deep learning. We talk about CNN in Chapter 3, which takes images or other matrices as input. We learn about how to deal with sequences in Chapter 4 and how to output a sequence in Chapter 5. Although many variants are not discussed, these can cover most of the applications.
We then see some advanced machine learning, as listed below.

Generative models: GAN, in Chapter 6;
Self-supervised learning: BERT, in Chapter 7 & 8;
What does a model learn: explainable ML, in Chapter 9;
Malice from humans: attack and defense, in Chapter 10;
Train and test are different: Domain Adaptation, in Chapter 11;
Learn from interaction and reward: RL, in Chapter 12;
The road to Skynet: life-long learning, in Chapter 13;
Edge computing: Network Compression, in Chapter 14;
Learn to learn: Meta Learning, in Chapter 15.

We see many applications. In terms of computer vision (CV), we see attack, adaptation, compression, explanation, anomaly detection, as well as generation. In terms of natural language processing (NLP), we see translation and question-answering. Moreover, we also see speech processing and playing games.

This is a long journey, and it's just the beginning, not the end. We just take a "one-term tour" in deep learning, aiming to show what technologies there are.

What to learn next?

After this course, you may find a problem you care about and try to solve it. You have the capability to read the papers and learn by yourself (NeurIPS, ICLR, AAAI, ICML, etc.)

Afterword

I never imagined writing a note up to 20,000 words when I started in the summer of 2023. However, the topic of ML is so magnificent that this note is simply a drop in the ocean. (In fact, Prof. Lee has several other courses similar to ML 2021, e.g., more detailed ML 2017, ML 2022 which is supplementary to ML 2021, ML 2023 which focuses more on generative models,

Introduction to Generative AI 2024 that is the latest, etc., let alone those various courses taught by other researchers.) Anyway, I try to mark down all the important points of ML 2021, and I hope that this note can serve as a dictionary for whoever has an interest in ML.

I am an undergraduate and do not major in AI. However, I try my best to write this note, and I really love the idea behind machine learning and want to learn more and more about it. I believe that one day, machine learning will no longer be magic, but a powerful tool for every one of us. I will be really glad if my note can help you.

Luke Olivaw

12 March, 2024@Shanghai