Deep Learning

Shuhei Watanabe

March 31, 2021

1 Introduction

Applications and research of deep learning (DL) are currently active thanks to **abundant libraries**, which allow non experts to implement algorithms without deep knowledge, and achieve **quick and strong performance** and **wide range of application** such as to image, video, graph data and so on. Additionally, DL can find good and abstract representation of given data [1]. Basically, DL learns such representations through the learning process. In the conventional ML, the domain experts extract such features or better representation of data. On the other hand, DL allows users to omit this process and realizes a completed pipeline of ML process, i.e. end-to-end learning. In the history of DL, there are two big waves. However, each of them declined for the following reasons:

1. Linear models cannot represent the XOR function and backlash against biologically inspired learning
2. Researchers could not realize big networks because of the lack of big data and adequate computational resources

At the first wave, logistic regression was invented, but this model [2] cannot represent the XOR function. At the second wave, those problems were resolved, but there were not sufficient computational power. However, the large computational server is now accecible for anyone and that led to the current third wave.

2 KL divergence vs Maximum likelihood estimation

The goal of DL is to approximate the distribution of the output $\boldsymbol{y}$ given a data point $\boldsymbol{x}$:

$$\min_{\boldsymbol{w}} \text{ discrepancy}(\hat{p}(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{w}), p(\boldsymbol{y}|\boldsymbol{x})).$$

It is achieved by the minimization of **KL divergence** between the modeled distribution and the target distribution as follows:

$$D_{\mathrm{KL}}(p\|\hat{p}) = \int p(\boldsymbol{y}|\boldsymbol{x}) \log \frac{p(\boldsymbol{y}|\boldsymbol{x})}{\hat{p}(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{w})} d\boldsymbol{y}.$$

Note that this metric is **not symmetric**, i.e. $D_{\mathrm{KL}}(p\|\hat{p}) \neq D_{\mathrm{KL}}(\hat{p}\|p)$ and the divergence is equal to zero if and only if $p = \hat{p}$ for all $\boldsymbol{y}$. In fact, the minimization of this metric is equivalent to the minimization of the cross entropy. To validate this statement, let's look at both formal and emprical forms of the cross entropy:

$$\begin{aligned} H(p,\hat{p}) &= -\int p(\boldsymbol{y}|\boldsymbol{x}) \log \hat{p}(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{w}) d\boldsymbol{y} \\ &\simeq -\frac{1}{N}\sum_{i=1}^{N} \log \hat{p}(\boldsymbol{y}_i|\boldsymbol{x}_i;\boldsymbol{w}) \end{aligned}$$

[1] For example, the roman numeral system is not a good representation for computation, but DL can extract better representation for this.

[2] Logistic regression can represent NAND, OR and AND. XOR is (NAND) AND (OR).

where $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ for all i $(1 \leq i \leq N)$ is the observation. Then, the conventional information entropy is the following:

$$\begin{aligned} H(p) &= -\int p(\boldsymbol{y}|\boldsymbol{x}) \log p(\boldsymbol{y}|\boldsymbol{x}) d\boldsymbol{y} \\ &\simeq -\frac{1}{N} \sum_{i=1}^{N} \log p(\boldsymbol{y}_i|\boldsymbol{x}_i). \end{aligned}$$

By combining both entropy, we will obtain the following:

$$D_{\mathrm{KL}}(p\|\hat{p}) = H(p, \hat{p}) - H(p).$$

The goal of DL is to identify the weight vector $\boldsymbol{w}$, which minimizes the cross entropy $H(p, \hat{p})$, and this equation can be reformulated as follows:

$$\boldsymbol{w}^\star \in \operatorname*{argmin}_{\boldsymbol{w}} H(p(\boldsymbol{y}|\boldsymbol{x}; \mathcal{D}), \hat{p}(\boldsymbol{y}|\boldsymbol{x}; \boldsymbol{w}))$$

where $\mathcal{D}$ is a given dataset. Since the posterior distribution $p(\boldsymbol{y}|\boldsymbol{x}; \mathcal{D})$ given a dataset $\mathcal{D}$ is independent from $\boldsymbol{w}$, the following holds:

$$\operatorname*{argmin}_{\boldsymbol{w}} H(p, \hat{p}) = \operatorname*{argmin}_{\boldsymbol{w}} (H(p, \hat{p}) - H(p)) = \operatorname*{argmin}_{\boldsymbol{w}} D_{\mathrm{KL}}(p\|\hat{p})$$

where $p = p(\boldsymbol{y}|\boldsymbol{x}; \mathcal{D}), \hat{p} = \hat{p}(\boldsymbol{y}|\boldsymbol{x}; \boldsymbol{w})$. By transforming the equation, we will obtain the following result:

$$\begin{aligned} \boldsymbol{w}^\star &= \operatorname*{argmin}_{\boldsymbol{w}} H(p, \hat{p}) = \operatorname*{argmin}_{\boldsymbol{w}} -p \ \log \hat{p} = \operatorname*{argmax}_{\boldsymbol{w}} p \ \log \hat{p} \\ &= \operatorname*{argmax}_{\boldsymbol{w}} \sum_{i=1}^{n} \log \hat{p}(\boldsymbol{y}_i|\boldsymbol{x}_i; \boldsymbol{w}) = \operatorname*{argmax}_{\boldsymbol{w}} \prod_{i=1}^{N} \hat{p}(\boldsymbol{y}_i|\boldsymbol{x}_i; \boldsymbol{w}). \end{aligned}$$

From this result, the maximum likelihood estimation (MLE) is proven to be equivalent to the minimization of KL divergence. KL divergence gives us zero if distribution is reproduced regardless of if the distribution has noise or not; therefore, KL divergence is the interest of observation in some contexts. Note that both formulations give negative infinity loss for wrong answers with high confidence and strictly punish them.

3 Neural networks

3.1 Non-linear regression vs neural networks

Neural networks learn the non-linear feature space through the learning process and **users do not have to define such non-linear mappings** so much. While each layer computes the weighted sum of the values from the previous layer, each node applies non-linear mapping and each layer stacks on such outputs. In the end, neural networks obtain the non-linear feature space suitable for a given task. On the other hand, non-linear regression takes the feature space by selecting the basis functions and these functions are basically not parameterized. While neural networks automatically learn the feature space, users have to design a suitable feature space by hand in non-linear regression.

3.2 Forward pass

Each layer of neural networks computes the weighted sum of the values from the previous layer and the activation function of the weighted sum. Let B be the batch size of an input vector, $m^{(l)}$ be the number of units in the l-th hidden layer and $a^{(l)}(\cdot)$ be the activation function at the $(l+1)$-th layer. Additionally, let the output from the l-th hidden layer be $\boldsymbol{Z}^{(l)} \in \mathbb{R}^{B \times D}$, the weight vector from the l-th

Table 1: The main usage of the activation function

Task	Activation functions for output layer
Regression	Linear
Binary classification	Logistic or tanh
Multi-class classification	softmax

to $(l+1)$-th layer [3] be $\boldsymbol{W}^{(l)} \in \mathbb{R}^{m^{(l)} \times m^{(l+1)}}$, the bias vector be $\boldsymbol{b}^{(l)} \in \mathbb{R}^{m^{(l+1)}}$. Then, the forward pass of neural networks is $\boldsymbol{Z}^{(l+1)} = a^{(l)}(\boldsymbol{Z}^{(l)})\boldsymbol{W}^{(l)} + \boldsymbol{B}^{(l)} = \boldsymbol{A}^{(l)}\boldsymbol{W}^{(l)} + \boldsymbol{B}^{(l)}$ where $\boldsymbol{B}^{(l)}$ is the broadcasted $\boldsymbol{b}^{(l)}$ defined as follows:

$$\boldsymbol{B}^{(l)} = \begin{bmatrix} (\boldsymbol{b}^{(l)})^\top \\ (\boldsymbol{b}^{(l)})^\top \\ \vdots \\ (\boldsymbol{b}^{(l)})^\top \end{bmatrix}.$$

The computational complexity and the memory complexity of the forward pass in the l-th layer is $O(Bm^{(l)}m^{(l+1)})$ and $O(m^{(l)}m^{(l+1)})$, respectively [4]. If $m^{(l)}$ is identical for all l, then the total computational complexity and the total memory complexity of the forward pass is $O(Bm^2L)$ and $O(m^2L)$, respectively.

3.3 Activation functions

- **Sigmoid**: $f(x) = 1/(1+e^{-x})$: Bounded in $[0, 1]$ and useful for the probabilistical representation
- **Softmax**: $f(y = i|\boldsymbol{x}) = e^{x_i} / \sum_{j=1}^{K} e^{x_j}$: Bounded in $[0, 1]$ and used for multi-class classification
- **Tanh**: $f(x) = (e^x + e^{-x})/(e^x - e^{-x})$: Bounded in $[-1, 1]$ and symmetric and does not change the center of distribution, useful for **hidden layers**
- **Linear**: $f(x) = x$: Useful for regression output layer
- **ReLU**: $f(x) = \max(0, x)$: Widely used the most, non-differentiable, but practically no problem. The advantages are the constant derivative values and the sparsity, but the problem is killed neurons are not updated at all.

Note that ReLU is widely used because it handles the **gradient vanishing problem** [5]. Other than those, Parametric ReLU (PReLU), Exponential Linear Unit (ELU), Gaussian Error Linear Unit, SWISH are introduced. Logistic or softmax function are used for the output layer in classification tasks because of **the differentiability**. We can use hard-max for the forward pass, but we need the differentiability in the backward pass. Table 1 shows the main use of the activation functions. When dealing with regression tasks, the linear function is preferred because it can represent the whole real numbers. On the other hand, for example, ReLU cannot represent the negative number, so it is not suitable for such regression tasks.

3.4 Representational capacity of deep learning

If Multi-layer perceptron (MLP) has more than two layers, the following theorem (**Universal Approximation Theorem**) holds:

[3] the 0-th layer is the input layer and the $(L+1)$-th layer is the output layer.

[4] In the backward pass, the memory complexity is $O(m^{(l)}m^{(l+1)} + Bm^{(l)} + Bm^{(l+1)})$

[5] For example, the gradients of sigmoid and tanh are smaller than or equal to 0.25 and 1, respectively. As the architecture becomes deeper, it will be sufferred from the gradient vanishing and slower convergence.

Theorem 1

1. *Any Boolean function, and*
2. *Any bounded continuous function*

can be realized or approximated with arbitrary precision by an MLP with one hidden layer.

However, we have to keep in mind the following notes:

1. The arbitrary function can be approximated if the **infinite** number of units are available, and
2. this theorem does not say that such MLP can be obtained, but it says there exists such MLPs.

Therefore, we need more advanced models. Recent research showed that DL has more flexible and sophisticated representational capacity. The Corollary 5 in [13] supports this proposition:

Theorem 2
A neural network with n_0 inputs and L layers of n units each, with ReLU activations can represent functions that have $\Omega((n/n_0)^{n_0(L-1)}n^{n_0})$ linear regions.

Roughly speaking, current ML models with ReLU draw decision boundary or estimated functions curve using the joint of linear lines. If the length of each linear region goes to zero, such a curve or a decision boundary becomes smoother and more flexible. Therefore, the increase in the order of linear regions leads to more flexible and powerful models. Since the exponential part of Theorem 2 only has the number of layers, but not the number of units, we can tell DL has more representational capacity than shallower or wider models.

3.5 Backward pass

3.5.1 Naïve computation

The following chain rule is required to compute backward pass.

Theorem 3
Given $x \in \mathbb{R}^m, y \in \mathbb{R}^n, f : \mathbb{R}^m \to \mathbb{R}^n, g : \mathbb{R}^n \to \mathbb{R}$ and let z be $g(y)$ and y be $f(x)$, then the following equation holds:

$$\frac{\partial z}{\partial x_j} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x_j}.$$

Let $X \in \mathbb{R}^{B \times D}$ be the input data, $Y \in \mathbb{R}^{B \times C}$ be the output label data (each row has onehot vector) and the batch size is B as in the forward pass section. Then, the forward pass is $\boldsymbol{Z}^{(l+1)} = \boldsymbol{A}^{(l)}\boldsymbol{W}^{(l)} + \boldsymbol{B}^{(l)}$.

In the same vein, we introduce the backward pass as follows:

$$\begin{aligned}
\frac{\partial L}{\partial \boldsymbol{A}^{(l)}} &= \frac{\partial L}{\partial \boldsymbol{Z}^{(l+1)}} \frac{\partial \boldsymbol{Z}^{(l+1)}}{\partial \boldsymbol{A}^{(l)}}, \\
\frac{\partial L}{\partial \boldsymbol{W}^{(l)}} &= \frac{\partial L}{\partial \boldsymbol{Z}^{(l+1)}} \frac{\partial \boldsymbol{Z}^{(l+1)}}{\partial \boldsymbol{W}^{(l)}}, \\
\frac{\partial L}{\partial \boldsymbol{b}^{(l)}} &= \frac{\partial L}{\partial \boldsymbol{Z}^{(l+1)}} \frac{\partial \boldsymbol{Z}^{(l+1)}}{\partial \boldsymbol{b}^{(l)}}, \\
\frac{\partial L}{\partial \boldsymbol{Z}^{(l)}} &= \frac{\partial L}{\partial \boldsymbol{A}^{(l)}} \frac{\partial \boldsymbol{A}^{(l)}}{\partial \boldsymbol{Z}^{(l)}}.
\end{aligned}$$

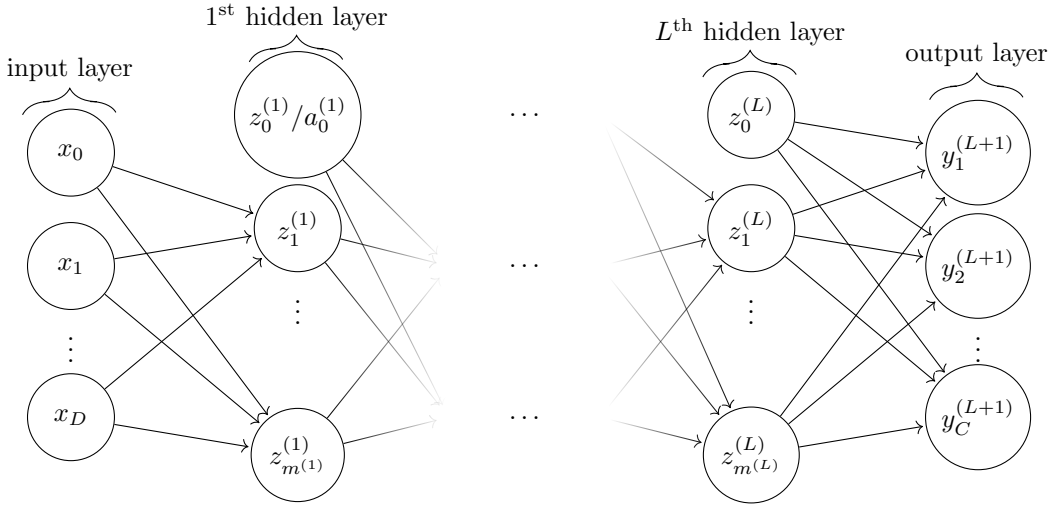

Figure 1: Network graph of a $(L+1)$-layer perceptron with D input units and C output units. The l-th hidden layer contains $m^{(l)}$ hidden units. $z_k^{(l)}$ is the weighted sum for the k-th node at the l-th hidden layer. The activation value for the corresponding node is $a_k^{(l)}$.

where $L = L(\boldsymbol{y}, \hat{\boldsymbol{y}})$ is the loss function. In this formulation, we have to store the 4 dimensional tensor and this is often infeasible.

3.6 Matrix computation of back propagation

Since $\boldsymbol{Z}_{i,j}^{(l+1)}$ is the value of the i-th batch at the of j-th unit, the gradients with respect to i'-th batch $\boldsymbol{A}_{i',:}^{(l)}$ and the weight to j'-th unit $\boldsymbol{W}_{:,j'}^{(l)}$ is zero $(i'(\neq i), j'(\neq j))$. Therefore, the computation of each element in $\boldsymbol{Z}^{(l+1)}$ is as follows:

$$\boldsymbol{Z}^{(l+1)} = \begin{bmatrix} z_{1,1}^{(l+1)} & \cdots & z_{1,m^{(l+1)}}^{(l+1)} \\ \vdots & \ddots & \vdots \\ z_{B,1}^{(l+1)} & \cdots & z_{B,m^{(l+1)}}^{(l+1)} \end{bmatrix} = \begin{bmatrix} b_1^{(l)} + \sum_{i=1}^{m^{(l)}} w_{i,1}^{(l)} a_{1,i}^{(l)} & \cdots & b_{m^{(l+1)}}^{(l)} + \sum_{i=1}^{m^{(l)}} w_{i,m^{(l+1)}}^{(l)} a_{1,i}^{(l)} \\ \vdots & \ddots & \vdots \\ b_1^{(l)} + \sum_{i=1}^{m^{(l)}} w_{i,1}^{(l)} a_{B,i}^{(l)} & \cdots & b_{m^{(l+1)}}^{(l)} + \sum_{i=1}^{m^{(l)}} w_{i,m^{(l+1)}}^{(l)} a_{B,i}^{(l)} \end{bmatrix}.$$

In the same vein, we do not have to compute most gradients and each element in matrices is the following:

$$\left(\frac{\partial L}{\partial \boldsymbol{A}^{(l)}}\right)_{i,j} = \sum_{k=1}^{m^{(l+1)}} \frac{\partial L}{\partial z_{i,k}^{(l+1)}} \frac{\partial z_{i,k}^{(l+1)}}{\partial a_{i,j}^{(l)}} = \sum_{k=1}^{m^{(l+1)}} \frac{\partial L}{\partial z_{i,k}^{(l+1)}} w_{j,k}^{(l)},$$

$$\left(\frac{\partial L}{\partial \boldsymbol{W}^{(l)}}\right)_{i,j} = \sum_{k=1}^{B} \frac{\partial L}{\partial z_{k,j}^{(l+1)}} \frac{\partial z_{k,j}^{(l+1)}}{\partial w_{i,j}^{(l)}} = \sum_{k=1}^{B} \frac{\partial L}{\partial z_{k,j}^{(l+1)}} a_{k,j}^{(l)},$$

$$\left(\frac{\partial L}{\partial \boldsymbol{b}^{(l)}}\right)_{i} = \sum_{k=1}^{B} \frac{\partial L}{\partial z_{k,i}^{(l+1)}} \frac{\partial z_{k,i}^{(l+1)}}{\partial b_{i}^{(l)}} = \sum_{k=1}^{B} \frac{\partial L}{\partial z_{k,i}^{(l+1)}}.$$

Since $a_{i,j}^{(l)}$ has connections to only the i-th data point at the following layer and $w_{i,j}^{(l)}$ has connections to only the j-th unit at the following layer, the chain rule has to be applied to only the aforementioned

elements. Therefore, the computation can be written in the following matrix forms:

$$\begin{aligned}\frac{\partial L}{\partial \boldsymbol{A}^{(l)}} &= \frac{\partial L}{\partial \boldsymbol{Z}^{(l+1)}}(\boldsymbol{W}^{(l)})^\top,\\ \frac{\partial L}{\partial \boldsymbol{W}^{(l)}} &= (\boldsymbol{A}^{(l)})^\top \frac{\partial L}{\partial \boldsymbol{Z}^{(l+1)}},\\ \frac{\partial L}{\partial \boldsymbol{b}^{(l)}} &= \left(\frac{\partial L}{\partial \boldsymbol{Z}^{(l+1)}}\right)^\top \mathbf{1}^B\end{aligned} \tag{1}$$

where $\mathbf{1}^B = [1, \cdots, 1]^\top \in \mathbb{R}^B$. Finally, $\partial L/\partial \boldsymbol{Z}^{(l)}$ is obtained using the chain rule as follows:

$$\frac{\partial L}{\partial \boldsymbol{Z}^{(l)}} = \frac{\partial L}{\partial \boldsymbol{A}^{(l)}} \odot \boldsymbol{G}^{(l)}$$

where $G^{(l)}_{i,j} = \partial a^{(l)}_{i,j}/\partial z^{(l)}_{i,j} = \partial a^{(l)}(z^{(l)}_{i,j})/\partial z^{(l)}_{i,j}$ and $\odot$ is the Hadamard product[6]. The initial gradients are the gradients of the output $\hat{Y} = \boldsymbol{A}^{(L+1)}$ with respect to $\boldsymbol{Z}^{(L+1)}$ as follows:

$$\frac{\partial L}{\partial \boldsymbol{Z}^{(L+1)}} = \frac{\partial L}{\partial \boldsymbol{A}^{(L+1)}}\frac{\partial \boldsymbol{A}^{(L+1)}}{\partial \boldsymbol{Z}^{(L+1)}} = \frac{\partial L}{\partial \boldsymbol{A}^{(L+1)}} \odot \boldsymbol{G}^{(L+1)}.$$

In the case where $g^{(L+1)}$ is the softmax, the gradients are the following:

$$\frac{\partial L}{\partial \boldsymbol{Z}^{(L+1)}} = \frac{\hat{Y} - Y}{N}.$$

By introducing the initial values to Eq. (1), we will obtain the whole gradient values. The overall memory complexity of back propagation is $O(LM^2 + LMB)$ in the case of $M = m^{(i)}$ for all $i = 1, \cdots L$.

4 Optimization for deep learning

DL requires the optimization of the weight tensor. The difficulty of optimization tasks depends on the properties of tasks. We start the discussion of convex optimization and turn to non-convex optimization. Finally, we introduce momentum, the adaptive learning rate and the initialization strategy of the weight vector.

4.1 Learning vs pure optimization

Learning has the following four differences compared to pure optimizations:

1. may have to use surrogate instead of the real objective such as softmax instead of hard-max,
2. optimization on partial data distribution instead of the real distribution,
3. often introduces regularization term to generalize, and
4. requires the summation over given data points, so computationally expensive for a large dataset

Note that surrogate may also enable better generalization.

4.2 Convex optimization

The definition of convex is as follows:

[6] Element-wise multiplication

Definition 1
Given a C^2 function $f(\boldsymbol{x}) : \mathbb{R}^D \to \mathbb{R}$, f is convex iff the Hessian matrix $H \in \mathbb{R}^{D \times D}$ is positive semidefinite such that $H_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j}$.

Note that Hessian matrix is symmetric if and only if the given function has $\frac{\partial^2 f(\boldsymbol{x})}{\partial \boldsymbol{x}_i \partial \boldsymbol{x}_j}$ for all pairs $(i, j), (i \neq j)$ and all of them are continuous. The definition of positive semidefinite is the following:

Definition 2
A matrix A is positive semidefinite if one of the following is satisfied:

1. $\forall \boldsymbol{x} \in \mathbb{R}^D$, $\boldsymbol{x}^\top A \boldsymbol{x} \geq 0$,
2. *the all eigenvalues are non-negative, or*
3. $\exists L \in \mathbb{R}^{D \times D}$ *s.t.* $A = LL^\top$.

The convex optimization has a convergence guarantee. However, there is an ill-condition, which makes the convergence slower. The ill-condition is defined based on the following condition number:

Definition 3
The condition number of the Hessian matrix H of a given convex function is $\kappa(H) = \frac{\lambda_{\max}}{\lambda_{\min}}$ where $\lambda_{\max}, \lambda_{\min}$ are the maximum and minimum eigenvalue, respectively.

When $\kappa(H)$ is large, the given function is called ill-conditioned and it causes zig-zag behaviors. There are two solutions for this.

1. **Momentum**: Introduce the decayed step size from the previous iterations, it suppresses the zig-zag behavior
2. **Preconditioning**: Transform the function into good condition and optimizes in the space. Then after optimizing in the space, it re-maps into the original space.

We will discuss the momentum [7] later and discuss the preconditioning first. To explain the method, let's take the quadratic example $f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^\top H \boldsymbol{x}$ where H is positive definite. In the preconditioning, we substitute this function with $g(\boldsymbol{y}) = f(P^{\frac{1}{2}}\boldsymbol{y})$ where P is an arbitrary positive definite symmetric matrix and $P^{\frac{1}{2}}$ is the square-root of P [8]. In the preconditioning, we optimize $g(\boldsymbol{y})$ with respect to $\boldsymbol{y}$. Since the Hessian matrix of $g(\boldsymbol{y})$ is $P^{\frac{1}{2}} H P^{\frac{1}{2}}$, we can change the condition number of g. Ideally, $P^{\frac{1}{2}} H P^{\frac{1}{2}}$ should be an identity matrix. For this reason, we would like to take P to satisfy this condition and such P is obtained as follows:

$$
\begin{aligned}
I = P^{\frac{1}{2}} H P^{\frac{1}{2}} &= P^{\frac{1}{2}} H^{\frac{1}{2}} H^{\frac{1}{2}} P^{\frac{1}{2}} = H^{-\frac{1}{2}} H^{\frac{1}{2}} H^{\frac{1}{2}} H^{-\frac{1}{2}} \\
(P^{\frac{1}{2}})^2 &= (H^{-\frac{1}{2}})^2 \\
P &= H^{-1}
\end{aligned}
$$

where we utilize the fact that H is positive definite and symmetric. In the case of non-convex optimization, there might not be H^{-1} and it does not mean the descent direction obtained from the Hessian matrix improves the function value even if we can compute H^{-1}.

[7] In gradient descent of the quadratic function, the convergence rate is $\|\boldsymbol{x}_{t+1} - \boldsymbol{x}^\star\| \leq \frac{\kappa(H)-1}{\kappa(H)-1} \|\boldsymbol{x}_t - \boldsymbol{x}^\star\|$. However, if it is with momentum, the convergence rate is $\|\boldsymbol{x}_{t+1} - \boldsymbol{x}^\star\| \leq \frac{\sqrt{\kappa(H)}-1}{\sqrt{\kappa(H)}-1} \|\boldsymbol{x}_t - \boldsymbol{x}^\star\|$.

[8] In the case of symmetric positive definite matrix, but not in the case of positive semidefinite matrix, Cholesky decomposition also works.

4.3 Optimization of neural networks

There are several distinctive properties in the optimization of neural networks.

1. **Non-convex**: Hessian cannot tell the best direction and no guarantee of global solutions.
2. **Large datasets**: It requires the summation over data points, so the complexity goes up
3. **High dimensionality**: The computation of Hessian is computationally expensive
4. **Structure of neural networks**: The response surfaces are likely to be harder

The loss function of neural networks are typically non-convex function and there are **many saddle points** [9], which are the attractors for second-order methods. Therefore, we cannot guarantee the convergence in the optimization. Furthermore, the following properties in the structure of neural networks make optimizations harder.

1. **Clif-like surfaces in the search space**: The gradients suddenly drop or rise significantly at some points especially in RNN.
2. **Gradient vanishing and exploding**: Since we have to multiply matrices again and again, each value is going to be very small or large by the power factor.

4.4 Stochastic gradient descent (SGD)

While the gradient descent handles **learning by epoch**, SGD computes loss values using a fraction of a given dataset. Note that learning by epoch is to learn all the training data points at the same time. Typically, the optimization by gradient descent is stable, but it takes a lot more time. Therefore, we use so-called **mini-batch**, i.e. **a part of the dataset**, for one update step. SGD stochastically picks a batch of data points and learns the batch at each step. Note that both methods see all the data points in 1 iteration and this iteration is called **epoch**. Mini-batch learning is preferred due to the following reasons:

1. Much quicker training allows much more iteration
2. Since the standard error of empirical loss function against true loss function decreases by the order of $O(\frac{1}{\sqrt{B}})$ where B is the batch size, the larger batch size does not make a big difference.
3. The training dataset can include many similar data points, so mini-batch learning is likely to yield the similar result to that of the full-batch training.
4. Since mini-batch includes less data points and the standard error is larger, the neural networks are **not likely to overfit** the statistics of a given data distribution.

The standard error is, roughly speaking, the expectable error of the distribution of a bootstrapped dataset from the true distribution and the empirical loss approaches the loss using full data as B goes to the number of data points. Note that when we use smaller batch size, we have to decrease the learning rate as well. This is because the smaller batch size has the same effect as the larger learning rate [31].

[9] When the loss goes down, it is more likely to have local minima and less likely to have saddle points.

4.5 Learning rate and momentum

4.5.1 Learning rate

It is known that the convergence (including local minima) of SGD is guaranteed when we set the learning rate α_t scheduling as follows:

$$\sum_{t=1}^{\infty} \alpha_t = \infty,$$
$$\sum_{t=1}^{\infty} \alpha_t^2 < \infty.$$

In this sense, we should schedule the learning rate using some methods and the following achieves such scheduling:

1. **Linear decay**: $\alpha_t = (1 - \frac{t}{\tau})\alpha_0 + \frac{t}{\tau}\alpha_\tau$ (τ is the final epoch),
2. **Exponential decay**: $\alpha_t = c^t \alpha_0$,
3. **Step decay**: decay the learning rate by a certain interval, and
4. **Cosine decay**: $\alpha_t = \frac{\alpha_0}{2}(1 + \cos\frac{t\pi}{\tau})$.

Note that we conventionally adapt the learning rate by checking the validation and training learning curve of NN.

4.5.2 Momentum

The momentum has been introduced to suppress the zigzag behavior [10] and it is formulated as follows:

$$\begin{aligned}
\boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \alpha\hat{\boldsymbol{g}}_t + \beta \underbrace{(\boldsymbol{w}_t - \boldsymbol{w}_{t-1})}_{\text{Let it be } \Delta\boldsymbol{w}_{t-1}}, \\
\boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \alpha\hat{\boldsymbol{g}}_t + \beta(-\alpha\hat{\boldsymbol{g}}_{t-1} + \beta\Delta\boldsymbol{w}_{t-2}), \\
&\dots \\
\boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \alpha\sum_{i=0}^{t}\beta^{k-i}\hat{\boldsymbol{g}}_i.
\end{aligned}$$

By transforming the equation, we obtain the following:

$$\begin{aligned}
\boldsymbol{G} &= \beta\boldsymbol{G} + \alpha\hat{\boldsymbol{g}}, \\
\boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \boldsymbol{G}.
\end{aligned}$$

The advantages of the momentum are **the smoothing effect of zig-zagging, the acceleration of learning at flat spots and the deacceleration of the learning when the sign of derivatives change**. On the other hand, the disadvantages of momentum are the additional parameter β and it potentially trigers additional zig-zagging when we use a strong momentum.

4.6 Gradient descent with adaptive learning rate

As mentioned previously, the learning rate has to be adapted for the learning convergence and different parameters (or dimensions) typically require a different learning rate. Therefore, **AdaGrad** has been

[10] Since the momentum takes over the dynamics from the previous iteration, it cancels out the force to go to the opposite direction.

invented [6]. The update of AdaGrad is as follows:

$$\begin{aligned}\boldsymbol{G} &= \boldsymbol{G} + \hat{\boldsymbol{g}} \odot \hat{\boldsymbol{g}},\\ \boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \frac{\alpha}{\sqrt{\boldsymbol{G}} + \epsilon} \odot \hat{\boldsymbol{g}}.\end{aligned}$$

where $\epsilon \ll 1$ is a positive number. When the accumulated gradients $\boldsymbol{G}$ is large, it implies the corresponding weight values have changed much compared to those have smaller $\boldsymbol{G}$. Therefore, AdaGrad slows down the learning by making the learning rate smaller. The problem of AdaGrad is the long term influences of the initial gradients and it potentially leads to premature learning. The revised version of AdaGrad is **RMSProp**. The update of this equation is the following:

$$\begin{aligned}\boldsymbol{G} &= \rho\boldsymbol{G} + (1-\rho)\hat{\boldsymbol{g}} \odot \hat{\boldsymbol{g}},\\ \boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \frac{\alpha}{\sqrt{\boldsymbol{G}} + \epsilon} \odot \hat{\boldsymbol{g}}.\end{aligned}$$

By introducing the decay rate ρ, RMSProp can reduce the influences from older epochs. Finally, we introduce **Adam** [16].

$$\begin{aligned}\boldsymbol{G}_1 &= \rho_1\boldsymbol{G}_1 + (1-\rho_1)\hat{\boldsymbol{g}},\\ \boldsymbol{G}_2 &= \rho_2\boldsymbol{G}_2 + (1-\rho_2)\hat{\boldsymbol{g}} \odot \hat{\boldsymbol{g}},\\ \hat{\boldsymbol{G}}_1 &= \frac{\boldsymbol{G}_1}{1-\rho_1^t},\\ \hat{\boldsymbol{G}}_2 &= \frac{\boldsymbol{G}_2}{1-\rho_2^t},\\ \boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \frac{\alpha}{\sqrt{\hat{\boldsymbol{G}}_2} + \epsilon} \odot \hat{\boldsymbol{G}}_1.\end{aligned}$$

The correction is based on the equation $\sum_{i=1}^{t} \rho^{t-i}(1-\rho) = (1-\rho)\frac{1-\rho^t}{1-\rho} = 1-\rho^t$. Note that the first moment $\boldsymbol{G}_1$ is equivalent to momentum. Although these methods adaptively change learning rate, it does not mean we do not have to schedule the learning rate. In fact,the learning rate scheduling can improve the performance of Adam [23].

4.7 The initialization of weights

Basically, weights are initialized using uniform or normal distribution [11] and bias is initialized using a constant number. In practice, the scale of the initialization is important. When the scale is larger, break symmetry [12] happens. On the other hand, if the scale is smaller, the training result would be similar to each other and thus it can be generalized. In the same way, when the scale is larger, we can make sure most activation functions do not output 0 in the forward pass. However, if we take smaller scale, we can avoid the gradient explosion.

Since we use the same learning rate for each layer, we cannot expect the same learning for each layer and thus the learning becomes difficult when we do not have **the same scales** in the whole architecture For this reason, the initialization has to be implemented in each layer separately and one effective strategy for this would be to check the scale of activation function values in each layer using mini-batch data and adapt the scale to make sure the even initialization for each layer. For the initialization of bias, there are some exceptional cases where we do not initialize by zero:

1. **The bias before ReLU unit**: We set a positive number (e.g. 0.1) to avoid the zero output
2. **Output units**: If we have prior information regarding the task, we put this information explicitly

[11] A well-known example is Xavier. $w \sim \mathcal{U}\left(-\sqrt{\frac{6}{d_{\text{in}}+d_{\text{out}}}}, \sqrt{\frac{6}{d_{\text{in}}+d_{\text{out}}}}\right)$

[12] When we set all the weights as an identical number, all the weights will be updated in the same way. Break symmetry is to break this phenomenon.

5 Regularization

The main idea behind the regularization is to reduce the variance in the expected test loss (= bias^2 + variance + noise) in exchange of some bias. One of the most naïve methods for the regularization is **bagging**. Bagging is to bootstrap samples from a training dataset and ensemble the trained models on each bootstrapped dataset, but this is not used in DL due to the expensive training cost.

5.1 Parameter sharing

If we can share parameters in a network over multiple tasks, we implicitly have more data for training the model. By the parameter sharing, we can achieve the following:

1. Since dealing with multiple tasks is harder, it can control the network's capacity
2. Since it shares parameters among similar tasks, it encourages the search for regular patterns

There are two major methods to achieve parameter sharing. One is to **share the shallower parts of parameters over multiple tasks** and since it implicitly increases the amount of training data, it can reduce both memory storage and computational time. The other is called **parameter tying** and it makes parameters of given two networks similar using a regularization term between their parameters as follows:

$$\Omega(\boldsymbol{w}^{(A)}, \boldsymbol{w}^{(B)}) = \|\boldsymbol{w}^{(A)} - \boldsymbol{w}^{(B)}\|_2$$

where $\boldsymbol{w}^{(A)}, \boldsymbol{w}^{(B)}$ are the weight parameters for task A and B.

5.2 Dropout

Dropout is to drop some weights in a given neural network model with a certain probability p during training. The basic concept is to discourage the co-adaptation of weights. In other words, dropout prevents the neural network model from relying on some weights too much and encourages each weight to have some more contributions to solving given tasks. Additionally, since the dropped graphs are the subgraphs of a given model, dropout can implicitly realize an ensemble. The important point is that we have to **store the location of dropped weights in the forward pass to ignore the update in the backward pass**. Note that since dropout drops some weights in the neural network model, we have to scale the output by considering the dropped weights. The right solution for this is the following:

$$f(\boldsymbol{x}; \boldsymbol{w}) = \sum_{\boldsymbol{w}_s \in \boldsymbol{w}} p(\boldsymbol{w}_s) f(\boldsymbol{x}; \boldsymbol{w}_s). \tag{2}$$

However, this is not tractable, so it is performed by dividing each output by $\frac{1}{1-p}$ in practice.

5.3 Data augmentation / Noise robustness

Data augmentation is to increase the data amount [13] by applying some simple transformation to a training dataset and add this transformed data to the training dataset. Since deep learning typically requires so much data to yield high performance and it is cumbersome to collect more data, data augmentation is an effective solution. The following are the concrete examples:

1. **Translation**: To move data along each axis
2. **Scaling**: To change the size scale of data
3. **Reflection**: To flip the left and right side of data [14]

[13] We usually augment only training dataset, but people sometimes augment test or validation dataset as well for the sake of measuring robustness.

[14] It does not make sense to augment by reflection in the case of letter recognitions.

4. **Rotation**: To tilt data
5. **Stretching**: To change the aspect ratio of data

Note that since **CNNs have invariance with respect to the movement along each axis, translation does not make sense.** The drawback of data augmentation is to require more computational time proportional to the data size. Additionally, the improvement diminishes as the augmented amount increases. For this reason, we do not increase more than 100x in practice.

Another option is to add random noise to various components of neural networks as follows:

1. **Inputs**: data augmentation
2. **Hidden units**: data augmentation of extracted features
3. **Weights**: Bayesian neural networks $p(f(\boldsymbol{x})|\boldsymbol{x}, \mathcal{D}) = \int p(f(\boldsymbol{x})|\boldsymbol{x}, \boldsymbol{w})p(\boldsymbol{w}|\mathcal{D})d\boldsymbol{w}$
4. **Outputs**: label smoothing

Label smoothing is to introduce some noise to labels. For example, suppose we would like to classify given images into 3 classes, the labels for this task looks like one-hot vector $\boldsymbol{y}_i = [0, 1, 0]$. However, by introducing noise, we can make it such as $\boldsymbol{y}_i = [0.1, 0.8, 0.1]$. This method stops the neural networks from overfitting to given specific classes. The drawback is to require the tuning of the noise control parameter.

5.4 Adversarial training

Adversarial data is data which decept trained models [12] and such data is synthesized by adding small noise to the direction which the loss increases. This method is called fast gradient sign method and formulated as follows:

$$\boldsymbol{x}' = \boldsymbol{x} + \epsilon \ \mathrm{sign}\left(\frac{\partial \mathcal{L}(y, \hat{y})}{\partial \boldsymbol{x}}\right) \tag{3}$$

where ϵ is a sufficiently small positive number. Basically, this new input $\boldsymbol{x}'$ increases the loss function. By adding adversarial data to training datasets, we can encourage local constancy[15] and mitigate sensitivity to perturbation in inputs.

5.5 Weight preference

First, it is ideal to stop trainings when validation loss reaches its minimum and starts to go up and **early-stopping** is the strategy stopping the training earlier. Another solution is **shrinkage method** or weight decay. This term is defined as the L^p-norm of the weight vector. The properties of this term depends on the parameter p. For example, $L1$-norm promotes sparsity and $L2$-norm solves colinearity. The update of the weight vector is performed as follows:

$$\begin{aligned}
\mathcal{L} &= \mathcal{L}_{\mathcal{D}} + \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2, \\
\boldsymbol{g} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = \boldsymbol{g}_{\mathcal{D}} + \lambda \boldsymbol{w}, \\
\boldsymbol{w} &= \boldsymbol{w} - \alpha(\boldsymbol{g}_{\mathcal{D}} + \lambda \boldsymbol{w}), \\
\boldsymbol{w} &= \underbrace{(1 - \alpha\lambda)}_{\text{Approaches 1}} \boldsymbol{w} - \alpha \boldsymbol{g}_{\mathcal{D}}
\end{aligned}$$

where α is a learning rate. In the case of the optimization methods with adaptive learning rate such as Adam and RMSProp, since the learning rate α for some weights goes down quickly, the weight decay

[15] The function is said to have local constancy if the output does not change much within a small region of space.

effect is small for the corresponding weights. In this sense, we have to adjust the regularization term by decouping as follows [23]:

$$\boldsymbol{w} = (1-\beta)\boldsymbol{w} - \alpha \boldsymbol{g}_{\mathcal{D}}$$

where β is a positive number. This trick allows the regularization effect to be active even when the learning rate goes down. Note that while SGD with the weight decouping is equivalent to SGD with L2 regularization, Adam with the weight decouping is not equivalent to Adam with L2 regularization. This is because the learning rate for each weight is different in Adam.

6 Convolutional neural networks (CNN)

6.1 Forward pass

CNNs are one type of neural networks and it performs well on data with grid-like structure such as image and time series. As real-world applications, it is applied to object detection, segmentation[16], image captioning, skeletal pose estimation. The forward pass of CNNs uses the operation called **cross-correlation**. Before introducing cross-correlation, we have to define the following **convolution**:

Definition 4
Given two matrices $\boldsymbol{A} \in \mathbb{R}^{H\times W}, \boldsymbol{K} \in \mathbb{R}^{H'\times W'}$ *, the convolution* $\mathbb{R}^{H'\times W'} \times \mathbb{R}^{H\times W} \to \mathbb{R}^{(H-H'+1)\times(W-W'+1)}$ *is the following operation:*

$$(\boldsymbol{K} * \boldsymbol{A})_{h,w} = \sum_{x=1}^{W'}\sum_{y=1}^{H'} K_{y,x} A_{h-y,w-x}$$

where $W'+1 \leq w \leq W+1, H'+1 \leq h \leq H+1$.

The cross-correlation is defined using convolution and fliped kernel $\boldsymbol{K}$ as follows:

$$\begin{aligned}(\boldsymbol{K} \otimes \boldsymbol{A})_{h,w} = (\boldsymbol{K}' * \boldsymbol{A})_{h+H'+1,w+W'+1} &= \sum_{x=1}^{W'}\sum_{y=1}^{H'} K'_{y,x} A_{h+H'-y+1,w+W'-x+1} \\ &= \sum_{x=1}^{W'}\sum_{y=1}^{H'} K_{H'-y+1,W'-x+1} A_{h+H'-y+1,w+W'-x+1} \\ &= \sum_{x=1}^{W'}\sum_{y=1}^{H'} K_{y,x} A_{h+y,w+x}\end{aligned} \tag{4}$$

where $\boldsymbol{K}'$ is 180-degree fliped kernel filter, i.e. $K'_{y,x} = K_{H'-y+1,W'-x+1}$ and $0 \leq h \leq H-H', 0 \leq w \leq W-W'$. **Cross-correlation is equivalent to the combination of fliped kernel and convolution** from Eq. (4). The example of fliped kernel is illustrated in Figure 2. The forward pass of CNNs are computed as follows:

$$Z_{B,c} = \sum_{j=1}^{C_{\text{in}}} K_{c,j} \otimes A_{B,j} + b_c \mathbf{1}_{H_{\text{out}}\times W_{\text{out}}}$$

where B, c are the index of batch and the output channel, $C_{\text{in}}, C_{\text{out}}$ are the number of channels in the current and the following layers and $H_{\text{out}}, W_{\text{out}}$ are the height and the width of the output image and $\boldsymbol{K} \in \mathbb{R}^{C_{\text{out}}\times C_{\text{in}}\times H'_{\text{in}}\times W'_{\text{in}}}$ is a kernel tensor and $\boldsymbol{A} \in \mathbb{R}^{B\times C_{\text{in}}\times H'_{\text{in}}\times W'_{\text{in}}}$ is a input value from the previous layer and $\boldsymbol{b} \in \mathbb{R}^{C_{\text{out}}}$ is a bias vector and $\boldsymbol{Z} \in \mathbb{R}^{B\times C_{\text{out}}\times H_{\text{out}}\times W_{\text{out}}}$ is the output of the layer. There are several components of CNNs.

[16] It detects objects in a pixel-wise way.

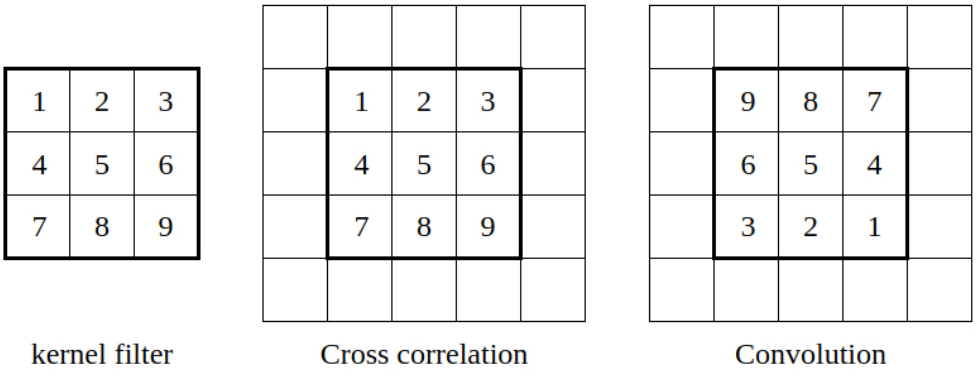

Figure 2: The conceptual visualization of cross correlation.

1. **Stride**: Change the step size of a convolution. For example, if a stride is set to 2, the filter will move two pixel at a time.
2. **Padding**: Pad the perimeter of image by some numbers, e.g. zero or the average of each pixel.
3. **Valid convolution**: Not perform the convolution if a filter is not within the target image.
4. **Same convolution**: Pad to keep the image size constant after the convolution.

Note that when a stride is S and a padding size P and the image size along one axis is X_{in} and the kernel size is K, then the output image size X_{out} along the axis is the following:

$$X_{\text{out}} = \left\lfloor \frac{X_{\text{in}} + 2P - K}{S} \right\rfloor + 1.$$

Additionally, there is a technique called **pooling**. Pooling is used to down-size an input image and it is invariant to small translations. Basically, **max**- or **average**-pooling is often used. For example, when we apply pooling to a given input, the formulation is the following:

$$\textbf{Max pooling}: \ A^{\text{out}}_{h,w} = \max_{1 \leq x \leq W', 1 \leq y \leq H'} A^{\text{in}}_{h+y,w+x},$$

$$\textbf{Average pooling}: \ A^{\text{out}}_{h,w} = \frac{1}{H'W'} \sum_{x=1}^{W'} \sum_{y=1}^{H'} A^{\text{in}}_{h+y,w+x}$$

where $0 \leq h \leq H - H', 0 \leq w \leq W - W'$. Note that pooling layers do not have any weight parameters and it is not common to use zero-padding for pooling layers.

6.2 Properties of CNNs

The CNNs extract abstract features from an input and the following convolution layer builds on top of the extracted features, so CNNs can learn more complex features. Additionally, there are the following three major properties in CNN:

1. **Sparse interactions**: Each element in a current layer interacts with limited number of units in the following layer and this leads to **the sparsity** and **less memory complexity** [17]. However, as the layers go deeper, each unit recepts much wider **receptive field**, because each element takes

[17] CNNs can reduce the memory complexity from $O(C_{\text{in}} C_{\text{out}} H_{\text{in}} W_{\text{in}} H_{\text{out}} W_{\text{out}})$ to $O(C_{\text{in}} C_{\text{out}} H'_{\text{in}} W'_{\text{in}})$.

over multiple features from the previous layer and the number of interacted elements increases exponentially with respect to the depth.

2. **Parameter sharing**: Since CNNs reuse the same weights multiple times in each layer at different regions and each parameter tries to capture regular patterns, it **reduces the memory complexity** and leads to the **generalization**, i.e. regularization.

3. **Equivariant representation**[18]: For example, when we define $f(x)$ as CNN and $g(x)$ as location shift. Since $f(g(x)) = g(f(x))$ holds, CNNs can yield the same result in both the prediction on shifted x, i.e. $f(g(x))$, and the shifted prediction on non-shifted x, i.e. $g(f(x))$. In other words, it has **translation invariant**. This property comes from weight sharing. Note that CNNs are not naturally equivariant to some other transformations such as changes in the scale or rotation of the image.

6.3 Miscellaneous convolutions

1. **3D convolution**: Normal convolution is 2D, but this convolution is expanded to 3D. It is useful when we handle data such as videos (height, width, time), MRI or CT images (height, width, depth)

2. **1×1 convolution**: It is used for the purpose of reducing channels and thus we can compute forward pass efficiently from the following layers. Since it extracts features and down-sizes data, it is called **feature pooling** and enables more complicated representation using non-linear activation after the procedure.

3. **Transposed convolution**: This convolution up-samples data by combining the padding and convolution.

4. **Dilated convolution**: It inflates the kernel by inserting spaces between kernel elements while keeping the kernel size; therefore no additional cost is required. Since it convolutes elements from a larger range, we can often get much larger receptive field especially when we use multi-dilated convolutions that are stuck together.

5. **Grouped convolution**: Filters at each layer are separated into certain numbers of groups and each group is responsible for convolutions of the corresponding group [19]. The advantages of this convolution are efficient training (each group can be allocated to different GPUs) and efficient model in terms of the number of model parameters [20].

6. **Spatially separable convolutions**: It first convolutes only along one axis and then it convolutes the convoluted data along the other axis. The adavantages of this convolution are less parameters and less matrix multiplications. On the other hand, training results can be sub-optimal.

7. **Depthwise separable convolutions**: It first convolutes each channel separately [21] , and then it applies 1×1 convolution to the data. This convolution also yields the same benefits and problems as spatially separable convolutions.

[18] The definition: $f(g(x)) = g(f(x))$

[19] Conventional CNNs convolute all the input channels.

[20] In the normal setting, the memory complexity of the kernel is $O(C_{\text{out}} C_{\text{in}} H'_{\text{in}} W'_{\text{in}})$. On the other hand, the kernel is $O(GC_{\text{out}}/GC_{\text{in}}/GH'_{\text{in}}W'_{\text{in}}) = O(C_{\text{out}}C_{\text{in}}H'_{\text{in}}W'_{\text{in}}/G)$ in the grouped convolution where G is the number of groups.

[21] More precisely, $Z_{B,c} = K_{c,1} \otimes A_{B,c}$ for all $c = 1, \cdots, C_{\text{in}}$. We do not take the summation over input channels.

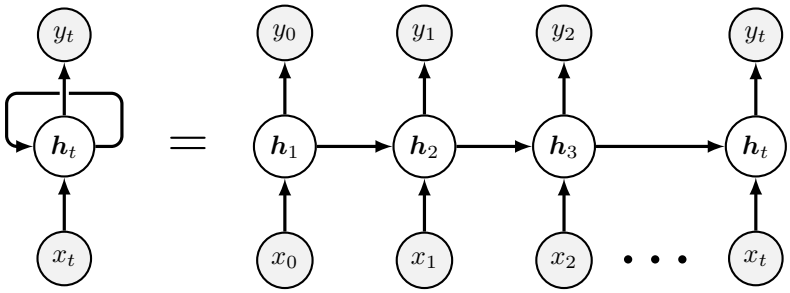

Figure 3: The visualization of RNN. The left figure is the short version and the right figure is the unfolded version.

7 Recurrent neural networks (RNN)

7.1 General settings

RNNs have a cyclic graph structure and typically skilled at dealing with sequential data, because it provides a time context and the concept of memory. RNNs implement dynamical systems rather than function mappings. Additionally, it is known as turing-complete [22]. There are several ways to use RNNs:

1. **One to many**: such as extracting the description (a sequence of words) of a given picture
2. **Many to one**: such as giving a score (one real value) to a given review (a sequence of words)
3. **Many to many**: such as translation from English (a sequence of words) to German (a sequence of words)

The forward pass of RNNs is the following:

$$\begin{aligned} \boldsymbol{h}_t &= \tanh(\boldsymbol{W}_h \boldsymbol{h}_{t-1} + \boldsymbol{W}_i \boldsymbol{x}_t + \boldsymbol{b}_h), \\ \hat{\boldsymbol{y}}_t &= g(\boldsymbol{o}_t) = g(\boldsymbol{b}_o + \boldsymbol{W}_o \boldsymbol{h}_t) \end{aligned} \tag{5}$$

where $\boldsymbol{h} \in \mathbb{R}^H$ is a hidden state vector, $\boldsymbol{x} \in \mathbb{R}^D$ is an input vector, $\boldsymbol{W}_h \in \mathbb{R}^{H \times H}, \boldsymbol{W}_i \in \mathbb{R}^{H \times D}, \boldsymbol{W}_o \in \mathbb{R}^{H \times K}$ are shared weight vectors for the hidden states, inputs and outputs, respectively, $\boldsymbol{b}_h \in \mathbb{R}^H, \boldsymbol{b}_o \in \mathbb{R}^K$ are shared bias vectors for the hidden states and outputs respectively, a is an activation function and K is the number of outputs for each time step. Note that the number of parameters does not depend on the sequence length and the memory complexity of the weight matrix is only $O(H(D+H))$.

7.2 Output-to-hidden connections and teacher forcing

When thinking about the back propagation through time (BPTT) [23] for Eq. (5), we need to compute all the gradients with respect to $\boldsymbol{h}_t$ using the information from the successive time steps. Therefore, it cannot be computed in parallel and it is more complicated process. One solution is to connect between the output $\boldsymbol{o}_t$ and the following hidden state $\boldsymbol{h}_{t+1}$ instead of $\boldsymbol{h}_t$ and $\boldsymbol{h}_{t+1}$. Since it lacks hidden-to-hidden recurrent connections, they are **likely to miss necessary information about the past history** of the input. On the other hand, the advantage is that all the gradients computation for each time step can be decoupled and that is why they can be computed in parallel. One variant is **teacher forcing**. Teacher forcing has connections from labels $\boldsymbol{y}_t$ to the successive hidden state $\boldsymbol{h}_{t+1}$ during training and from predictions $\boldsymbol{o}_t$ to $\boldsymbol{h}_{t+1}$ during test. Since this method uses the labels directly,

[22] Roughly speaking, if a given system can solve all the problems, which can be resolved by current computer systems, it is said to be turing-complete.

[23] BPTT is also weight sharing.

the **training is easier**. However, predictions must rely on predicted values during test and it is **likely to accumulate errors over time**.

7.3 Miscellaneous architectures

By transforming Eq. (5), we can easily obtain the formulations for one-to-many, many-to-many, many-to-one.

1. **Many to one**: Just limit the approximation to only at the last step to summarize the information and train by BPTT
2. **One to many**: Use same $\boldsymbol{x}$ for each time step $t = 1, 2, \cdots, T$
3. **Many to many**: Can be Combined with BPTT and teacher forcing

As other variants, there are **bidirectional RNNs** and **encoder-decoder sequence-to-sequence architectures**. Bidirectional RNNs have literally two directional hidden states. One is the forward direction and the other is the backward direction. The advantage is to be able to use information from both past and future. On the other hand, since it requires future information, we cannot use it for online tasks. Encoder-decoder sequence-to-sequence architectures are used when the length of inputs and outputs are different such as translation.

7.4 Back propagation

7.4.1 Computation

Suppose we would like to solve many-to-many tasks and we classify into K classes at each time step. Then the forward pass at time step t is the following:

$$
\begin{aligned}
\boldsymbol{h}_t &= \tanh(\boldsymbol{W}_h \boldsymbol{h}_{t-1} + \boldsymbol{W}_i \boldsymbol{x}_t + \boldsymbol{b}_h), \\
\boldsymbol{o}_t &= \boldsymbol{b}_o + \boldsymbol{W}_o \boldsymbol{h}_t, \\
\hat{\boldsymbol{y}}_t &= \mathrm{softmax}(\boldsymbol{o}_t), \\
\mathcal{L}_t &= \mathcal{L}(\hat{\boldsymbol{y}}_t, \boldsymbol{y}_t), \\
\mathcal{L} &= \sum_{t=1}^{T} \mathcal{L}_t.
\end{aligned}
$$

Therefore, the backward pass at time step t, which does not require the whole pass, is the following:

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathcal{L}_t} &= 1, \\
\frac{\partial \mathcal{L}_t}{\partial \boldsymbol{o}_t} &= \hat{\boldsymbol{y}}_t - \boldsymbol{y}_t, \\
\frac{\partial \mathcal{L}_t}{\partial \boldsymbol{W}_o} &= \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{o}_t} \frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{W}_o} = \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{o}_t} \boldsymbol{h}_t^\top \left(W_{o\ i,j} = \frac{\partial \mathcal{L}_t}{\partial o_{t,i}} h_{t,j} \right), \\
\frac{\partial \mathcal{L}_t}{\partial \boldsymbol{b}_o} &= \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{o}_t} \frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{b}_o} = \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{o}_t} \left(b_{o\ i} = \frac{\partial \mathcal{L}_t}{\partial o_{t,i}} \right).
\end{aligned}
$$

The gradients with respect to $\boldsymbol{h}_t, \boldsymbol{h}_{t-1}$ are the following:

$$
\frac{\partial L_t}{\partial \boldsymbol{h}_t} = \frac{\partial L_t}{\partial \boldsymbol{o}_t} \frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{h}_t} = \boldsymbol{W}_o^\top \frac{\partial L_t}{\partial \boldsymbol{o}_t} \left(\because \frac{\partial L_t}{\partial h_{t,i}} = \sum_{j=1}^{K} W_{o,j,i} o_{t,j} \right)
$$

$$
\frac{\partial L_t}{\partial \boldsymbol{h}_{t-1}} = \frac{\partial L_t}{\partial \boldsymbol{h}_t} \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}} = \boldsymbol{W}_h^\top \left(\frac{\partial L_t}{\partial \boldsymbol{h}_t} \odot (\boldsymbol{1} - \boldsymbol{h}_t^2) \right) \left(\because \frac{\partial L_t}{\partial \boldsymbol{h}_{t-1}} = \sum_{j=1}^{H} W_{h\ j,i} \frac{\partial L_t}{\partial h_{t,j}} (1 - h_{t,j}^2) \right)
$$

where $\frac{\partial y}{\partial x} = (1-y^2)w$ when $y = \tanh(wx+b)$. Using the gradients above, other gradients are computed as follows:

$$\frac{\partial \mathcal{L}_t}{\partial \boldsymbol{W}_h} = \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{h}_t}\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{W}_h} = \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{h}_t}(\boldsymbol{1} - \boldsymbol{h}_t^2)\boldsymbol{h}_{t-1} = \left(\frac{\partial \mathcal{L}_t}{\partial \boldsymbol{h}_t} \odot (\boldsymbol{1} - \boldsymbol{h}_t^2)\right)\boldsymbol{h}_{t-1}^\top \left(\because \frac{\partial \mathcal{L}_t}{\partial W_{h\ i,j}} = \frac{\partial \mathcal{L}_t}{\partial h_{t,i}}(1-h_{t,i}^2)h_{t-1,j}\right)$$

$$\frac{\partial \mathcal{L}_t}{\partial \boldsymbol{W}_i} = \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{h}_t}\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{W}_i} = \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{h}_t}(\boldsymbol{1} - \boldsymbol{h}_t^2)\boldsymbol{x}_t = \left(\frac{\partial \mathcal{L}_t}{\partial \boldsymbol{h}_t} \odot (\boldsymbol{1} - \boldsymbol{h}_t^2)\right)\boldsymbol{x}_t^\top \left(\because \frac{\partial \mathcal{L}_t}{\partial W_{i\ i,j}} = \frac{\partial \mathcal{L}_t}{\partial h_{t,i}}(1-h_{t,i}^2)x_{t,j}\right)$$

$$\frac{\partial \mathcal{L}_t}{\partial \boldsymbol{b}_h} = \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{h}_t}\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{b}_h} = \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{h}_t} \odot (\boldsymbol{1} - \boldsymbol{h}_t^2)^\top \left(\because \frac{\partial \mathcal{L}_t}{\partial b_{h,i}} = \frac{\partial \mathcal{L}_t}{\partial h_{t,i}}(1-h_{t,i}^2)\right).$$

In the whole pass, we use $\frac{\partial \mathcal{L}}{\partial :} = \sum_{t=1}^T \frac{\partial \mathcal{L}}{\partial \mathcal{L}_t}\frac{\partial \mathcal{L}_t}{\partial :}$ and the gradients of hidden states are the following:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_t} = \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{h}_t} + \frac{\partial \mathcal{L}_{t+1}}{\partial \boldsymbol{h}_t} = \boldsymbol{W}_o^\top \frac{\partial \mathcal{L}}{\partial \boldsymbol{o}_t} + \boldsymbol{W}_h^\top \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_{t+1}} \odot (\boldsymbol{1} - \boldsymbol{h}_{t+1}^2)\right).$$

After the whole computation of the gradients of loss with respect to hidden states, we can compute the gradients of each matrix as follows:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_h} &= \sum_{t=2}^T \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_t} \odot (\boldsymbol{1} - \boldsymbol{h}_t^2)\right)\boldsymbol{h}_{t-1}^\top, \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_i} &= \sum_{t=1}^T \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_t} \odot (\boldsymbol{1} - \boldsymbol{h}_t^2)\right)\boldsymbol{x}_t^\top, \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}_h} &= \sum_{t=1}^T \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_t} \odot (\boldsymbol{1} - \boldsymbol{h}_t^2)^\top.
\end{aligned}$$

7.4.2 Gradient vanishing and explosion

Although RNNs have powerful non-linear processing due to the repeating application of the same function, it also induces some problems. Typical problems are the gradient vanishing and explosion. This is due to the formulation of RNNs.

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_t} &\propto \boldsymbol{W}_h^\top \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_{t+1}} \odot (\boldsymbol{1} - \boldsymbol{h}_{t+1}^2)\right) \\
&\propto \boldsymbol{W}_h^\top \left(\boldsymbol{W}_h^\top \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_{t+2}} \odot (\boldsymbol{1} - \boldsymbol{h}_{t+2}^2)\right) \odot (\boldsymbol{1} - \boldsymbol{h}_{t+1}^2)\right) \\
&\dots \\
&\propto \left((\boldsymbol{W}_h^\top)^{T-t} \bigodot_{\tau=t}^{T} (\boldsymbol{1} - \boldsymbol{h}_\tau^2)\right)\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T}.
\end{aligned}$$

Let $\boldsymbol{W}_h = V\Lambda V^{-1}$ be the eigendecomposition of the weight matrix. In this case, the following holds:

$$\frac{\partial L}{\partial \boldsymbol{h}_t} \propto (1-h^2)^{T-t} V \Lambda^{T-t} V^{-1} \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T}. \tag{6}$$

Therefore, if the maximum or minimum of absolute eigenvalue is larger or smaller than 1, it leads to explosion or vanishing. The solution for the explosion is **gradient clipping** and that for the vanishing is **LSTM**.

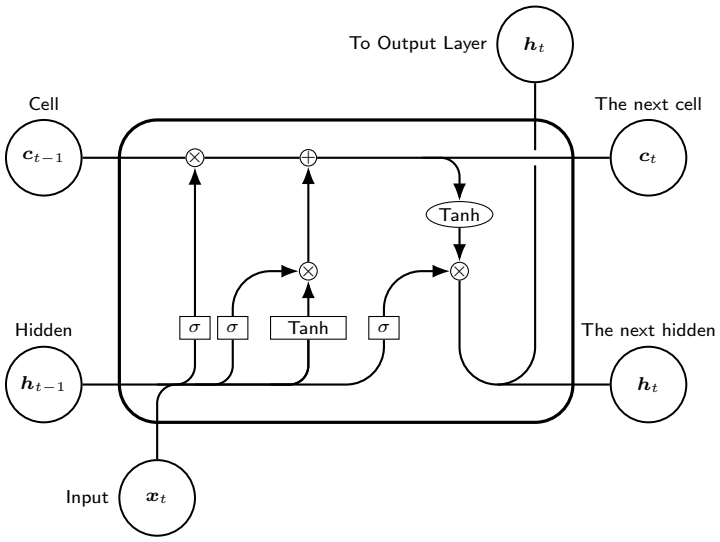

Figure 4: The visualization of LSTM.

7.5 Long Short-Term Memory (LSTM)

LSTM is invented as a solution for the vanishing gradient. This model is composed of three types of gates , i.e. **forget gate**, **input gate**, **output gate**, as visualized in Figure 4. Since this model has a self-loop of inner state $\boldsymbol{c}$ without any activations, the gradient with respect to $\boldsymbol{c}$ does not vanish nearly as RNNs do.

7.5.1 Structure

The formulation is as follows:

$$\begin{aligned}\begin{bmatrix}\boldsymbol{f}_t^\top & \boldsymbol{i}_t^\top & \tilde{\boldsymbol{c}}_t^\top & \boldsymbol{o}_t^\top\end{bmatrix}^\top &= \boldsymbol{W}_h\boldsymbol{h}_{t-1} + \boldsymbol{W}_i\boldsymbol{x}_t + \boldsymbol{b}_h,\\ \boldsymbol{c}_t &= \sigma(\boldsymbol{f}_t)\odot\boldsymbol{c}_{t-1} + \sigma(\boldsymbol{i}_t)\odot\tanh(\tilde{\boldsymbol{c}}_t),\\ \boldsymbol{h}_t &= \tanh(\boldsymbol{c}_t)\odot\sigma(\boldsymbol{o}_t),\\ \hat{\boldsymbol{y}}_t &= a(\boldsymbol{W}_o\boldsymbol{h}_t + \boldsymbol{b}_o).\end{aligned}$$

where $\boldsymbol{f}, \boldsymbol{i}, \tilde{\boldsymbol{c}}, \boldsymbol{o} \in \mathbb{R}^H$. Since the derivatives move from $\boldsymbol{c}_t$ to $\boldsymbol{f}_t, \boldsymbol{i}_t, \tilde{\boldsymbol{c}}_t$ and their derivatives move to $\boldsymbol{h}_{t-1}$ and $\frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{c}_{t-1}} = \sigma(\boldsymbol{f}_t)$, the decaying rate is far better than that of Eq. (6). The forget gate decays less important features from the past and maintain important features. The input gate takes two vectors $\boldsymbol{i}_t$ and $\tilde{\boldsymbol{c}}_t$ and they control how much new information we should store and in which direction we should change the state. The output gate controls which features to use. The advantages of LSTM are that it can **selectively forget and remember features** and **each parameter of LSTM already have their meanings**. For this reason, the training result can be easily interpreted and LSTM can yield good performance with less paramters and less training.

7.6 Alternatives

1. **Gated recurrent units (GRU)**: Since it combines the forgetting factor and the new contribution to the state update in one gate, it has slightly simpler and fewer parameters compared to LSTM.

2. **Echo state networks**: It uses a large fixed hidden part called reservoir with random weights and only trains output layer with linear regression. The idea is to use the rich connections and

rich dynamics in reservoir. Even though it is simple and quick, it shows strong performance. Note that we need to pay attention to the spectral radius [24].

3. **Neural turing machines**: This is an extension of RNN using the concept called memory cells. This method reads and writes the features of tasks in memory cells and this mechanism is called **attention**. It is skilled at dealing with tasks requiring long-term memories.

8 Practical methodology

8.1 Attention mechanism

Attention mechanism is to pay attention to more important features in given data rather than dealing with all the inputs equally. It is especially helpful to process big images and it reduces the computational complexity. For example, when we have images of 256×256 pixel size, but if they only have information in the center of the size 32×32, the attention can reduce the complexity significantly. Such attention mechanism is realized by weighting each pixel differently. There are mainly two types attentions to realize it: **Soft attention** and **hard attention**.

Soft attention first extracts feature using CNN and RNN learns weights for each feature at a given time step. Therefore, soft attention is used for image captioning. Since soft attention requires all the pixels and puts weights on each pixel, **it does not reduce the computational time**. For this reason, the hard attention has been introduced. Hard attention pays attention to only one location and since the sample of locations, i.e. the positions of cropping, cannot be learned using back propagation, such a location sampling is learned via **reinforcement learning** [25]. Hard attention is used for letter recognition. On the other hand, soft attention uses weighted sum of each feature from each location and **learns such weights using back propagation**. Note that there have been differentiable hard attention methods and one of them is **spatial transformer**. This method trains parameterized function that transforms inputs into cropped (therefore, smaller than the original inputs) inputs using back propagation and interpolates the transformed inputs using bilinear interpolation if required.

8.2 Detection and segmentation

Semantic segmentation is a pixel-wise labeling and CNNs for each pixel are trained in the most naïve setting. However, the pixel-wise training is computationally expensive, so recent research suggests to extract feature first while maintaining the image size and give the likelihoods in the final layer. Although it reduces runtime, we need to maintain the resolution of the original images. For this reason, we combine the feature extraction of CNNs and **transpose convolution**, i.e. upsampling.

Object detection is a combination of classification and localization. Since it has to optimize two tasks, it will optimize the multitask loss of softmax loss from a classification task and L2 loss from a localization task. Note that localization task is to minimize the error between prediction of the vertices of bounding box and the central position and the actual location of the bounding box. In the case of multiple objects, we apply the task to each sub-image chosen by the sliding window manner. However, this computation is expensive due to the huge candidate pool [26]. For this reason, Region CNN (RCNN) and You Only Look Once (YOLO) have been invented. RCNN learns region proposal network and searches from each region. YOLO is an end-to-end learning and first divides images into grid cells, then it approximates the shape of bounding box and the class label for each cell. Since grid cells are coarse, the accuracy is lower, but still achieves good performance.

As other applications, there are **instance segmentation** and **panoptic segmentation**. Instance segmentation not only classifies each pixel into a specific class, but also detects if each cell belongs to one specific object (e.g. human, car) in a given image. Panoptic segmentation is the combination of

[24] The largest absolute eigenvalues of connectivity matrix should be smaller than 1.
[25] Image is a state and a sample is an action. Based on the result, the sample will get a reward.
[26] Combinations of locations, scales, aspect ratio and so on.

Figure 5: The visualizations of semantic segmentation and object detection.

Figure 6: The visualizations of instance segmentation and panoptic detection.

instance segmentation and the regular segmentation. It applies instance segmentation to, for example, people or cars and uses regular segmentation for road and buildings.

8.3 CNN architectures

There are 10 types of famous architectures:

1. **LeNet5**: The first CNN model.
2. **AlexNet**: The CNN used in 2012.
3. **ZFNet**: A variant of AlexNet.
4. **VGGNet**: Small filter and deeper layers.
5. **GoogLeNet**: Inception module and less parameters. Inception module uses multiple kernel size convolution and concatenate all outputs depth-wisely.
6. **ResNet**: Residual block (a solution for the gradient vanishing) and deeper layer. Each residual block doubles the number of filters and halves pixel size. It also uses bottleneck layer (1×1 convolution layers)
7. **WideResNet**: It reduces the layers and widens channel size, so parallelizable.
8. **ResNeXt**: It parallelizes residual blocks.
9. **DenseNet**: It handles gradient vanishing by taking each input at each layer in successive layers and encourages feature reuse.
10. **MobileNet**: It uses depthwise separable convolution, so faster computation.
11. **SENet (Squeeze-and-Excitation Networks)**: A feature recalibration module that learns to adaptively reweight feature maps

In the previous layer of fully connected layer, CNN typically requires flatten operation and this size depends on the input image size. However, models such as GoogLeNet and ResNet uses global pooling before the layer, so we do not need to consider the pixel size anymore.

8.4 Batch normalization

Batch normalization is one of the **regularizations** and makes training easier thanks to the **handling of the internal covariate shift** and zero-mean of the activated values. Note that zero-mean of the activated values **prevent gradient vanishing/exploding** and allow higher learning rates[27]. Additionally, higher learning rates lead to **faster convergence**. Since batch normalization activates dead neurons, it improves gradient flow. Another benefit is **less sensitivity to the weight initialization**, because it normalizes values at each layer. The opeartion of batch normalization is performed by the following:

$$\mathrm{BN}(x) = \gamma \frac{x - \mu}{\sigma} + \beta,$$
$$\text{where } \mu = \frac{1}{B}\sum_{b=1}^{B} x_b, \sigma^2 = \frac{1}{B}\sum_{b=1}^{B}(x_b - \mu)^2$$

and β, γ are learnable parameters. When $\beta = \mu, \gamma = \sigma$, batch normalization becomes an identity function. In the case of MLP, each dimension is processed independently while each pixel is processed jointly, i.e. taking the mean and variance over the batch of whole images, in CNNs. Note that batch normalization is used after fully-connected or convolutional layer to undo the scale and before nonlinearity to reduce dead neurons. Additionally, during test, batch normalization uses μ and σ obtained by averaging values during training. For this reason, batch normalization exhibits different behaviors for training and testing.

The major problems of batch normalization are the handling of **mini-batch learning**, because of the unstable statistics and **recurrent neural structures**, because each time step has different statistics and requires another batch normalization layer. To address this issue, the following normalizations are invented:

1. **Batch normalization** (batch, pixel): Basic normalization. In the case of MLP, handles each dimension independently.
2. **Layer normalization** (pixel, all channels): Used for RNN.
3. **Instance normalization** (pixel): Used for style transfer and GAN.
4. **Group normalization** (pixel, some channels): Composing a group by some channels

Since all of them do not take statistics over batch, they behave in the identical way during both training and test. Note that all the normalization techniques take the normalization with respect to each dimension independently in MLPs.

8.5 Transfer learning

Transfer learning or fine-tuning is to use pre-trained weights as parts of neural networks for new tasks. Since the shallower parts of weights are more generic and the deeper parts of weights are more specific, we only initialize deeper layers and train only the deeper parts of networks. Practically, we use lower learning rate and train more layers for bigger dataset. Table 2 lists case-by-case solutions. Fine-tuning is **effective especially when large amount of data is not available**.

Another transfer learning is self-supervised learning and performed as follows:

1. Create pretext tasks using massive unlabeled data
2. Solve the pretext tasks
3. Evaluate the performance on a labeled dataset

[27] Since each weight will be normalized, the higher weight values are not likely to be a problem; therefore, we can take a higher learning rate.

Table 2: The case study of fine-tuning

Similarity / size	Little	Large
Similar	Train only final layer	Fine-tune a few layers
Not Similar	Fine-tune a larger number of layers	Try a linear layer from a shallower part

Note that the pretext tasks are tasks generated from unlabeled data easily. For example, the rotation detection can be generated easily, because we just need to apply rotation operation to each image and those operations will be the answers.

8.6 Debugging strategies

To quickly debug our model, we first train on a tiny dataset and keep modifying the model until reaching good performance. Another strategy is to monitor quantitative metrics such as loss function and visualize the inference on feeded data. When the training does not work well, one should monitor gradients and activations to know how much neurons are dead and compare the gradients with finite differential equation.

9 Hyperparameter optimization (HPO)

9.1 Practical tips

In HPO, there are mainly four types of parameters: **numerical**, **discrete**, **categorical** and **conditional** parameters. Besides, we need to consider the scale (e.g. **log and uniform**) of discrete and numerical parameters. This knowledge is important when designing the search space. When the model shows overfitting, we have to check **the capacity of the model**, **HPO methods** and **the degree of regularization**. On the other hand, when the model shows underfitting, we have to check **different regularizations**, **double descent phenomenon** [28] or **train on more data** if available.

In general, HPO is not useful as long as models cannot overfit training datasets. For this reason, one good strategy is to make sure that the model (i.e. training loss) can overfit the subset and then move to the HPO of validation loss on the full dataset. As another tip, the learning rate is typically important, so one heuristic is to increase learning rate until the loss diverges and reduce it a little bit to determine the domain for the learning rate.

9.2 Black-box optimization

Black-box optimization is to optimize functions without derivative information. The most basic algorithm is random search and it maintains global search and can be computed in parallel. Additionally, the random search can deal with low intrinsic dimensionality compared to grid search. Another famous example is **local search** and it fixes the best configuration and changes the configuration along only a single axis.

Those methods are naïve methods and not sample-efficient. Therefore, there are several methods using previous observations. One example is Bayesian optimization and it fits a probabilistic model $\mathcal{N}(\mu(\theta), \sigma^2(\theta))$ where θ is a configuration. Typical Bayesian optimization balances the trade-off between exploration and exploitation and finds the next configurations to be evaluated from the surrogate search space. Since Bayesian optimization itself is not cheap compared to random search, it is used when the objective function is expensive. One major example of Bayesian optimization is TPE and this

[28] The phenomenon where the loss goes down when increasing the model capacity and goes up at some points, but when we keep increasing the capacity, the loss goes down again.

method can deal with conditional and categorical parameters. Another example is a population-based method and it maintains diversity and improves the fitness of the whole population.

9.3 Speedup techniques of HPO

Since the objectives in HPO is usually expensive, many speedup techniques have been studied. In the lecture, meta-learning across datasets [11, 34], extrapolation of learning curves [4] and multi-fidelity optimization [14, 17, 19, 30] are introduced. Note that multi-fidelity optimization is to use cheap-to-evaluate proxy instead of the high-fidelity objective. The concrete examples are the early-stopping of poor performance and removing poor configurations based on the training on a tiny dataset.

9.4 Hyperparameter gradient descent

Recently, the gradient-based HPO method has been invented [24] and it is formulated as follows:

$$\min_{\theta} \mathcal{L}_{\text{val}}(\boldsymbol{w}^{\star}(\theta), \theta)$$
$$\text{where } \boldsymbol{w}^{\star}(\theta) \in \operatorname*{argmin}_{\boldsymbol{w}} \mathcal{L}_{\text{train}}(\boldsymbol{w}, \theta).$$

Note that the optimization of weights is called **inner objective** and that of hyperparameters is called **outer objective**. Although it requires twice much computation and there is no guarantee of the convergence, it might potentially handle more hyperparameters well. Additionally, the bi-level optimization allows quicker optimization and it updates weights at training step and hyperparameters at validation step.

10 Neural architecture search (NAS)

In this section, we will discuss how to find the optimal neural architecture. Note that since NAS requires substantial amount of computation and neural architecture does not make big difference compared to hyperparameter configurations, we recommend to apply HPO first unless we have adequate resources.

10.1 Search space

10.1.1 Basic search space

NAS optimizes the objective $f(a)$ where $a \in \mathcal{A}$ is architecture composed of **micro**, **macro** and **hierarchical** architectures [9]. The most basic search space is to align the components in the chain-structure. Another search space is to build a graph by choosing such components. However, since the selection of such components leads to hard combinatorial optimization problems, it is often infeasible. Therefore, there have been many papers introducing search spaces.

10.1.2 Cell search space

This space focuses on discovering the architecture of specific cells that can be combined to assemble the entire neural network [38]. The cells are made up of B blocks [29] (e.g. the connection of hidden layers and kernel size) and we obtain architectures by stacking such cells as shown in Figure 7. There are two types of cells. One is a normal cell that maintains the shape of inputs and the other is a reduction cell that recudes the shape of inputs. Since the searching of the optimal cell yields a smaller search space compared to architecture space, it leads to **quicker search**. Additionally, the cells can be **transferred to other tasks**. Although stacking repeating patterns are proven to be a useful design (e.g. ResNet), it can miss the possiblities that **the different cells work well in the different parts** of the architecture. Another drawback is that we need to determine the macro architecture.

[29] The number of hyperparameters is the choices of each block times B.

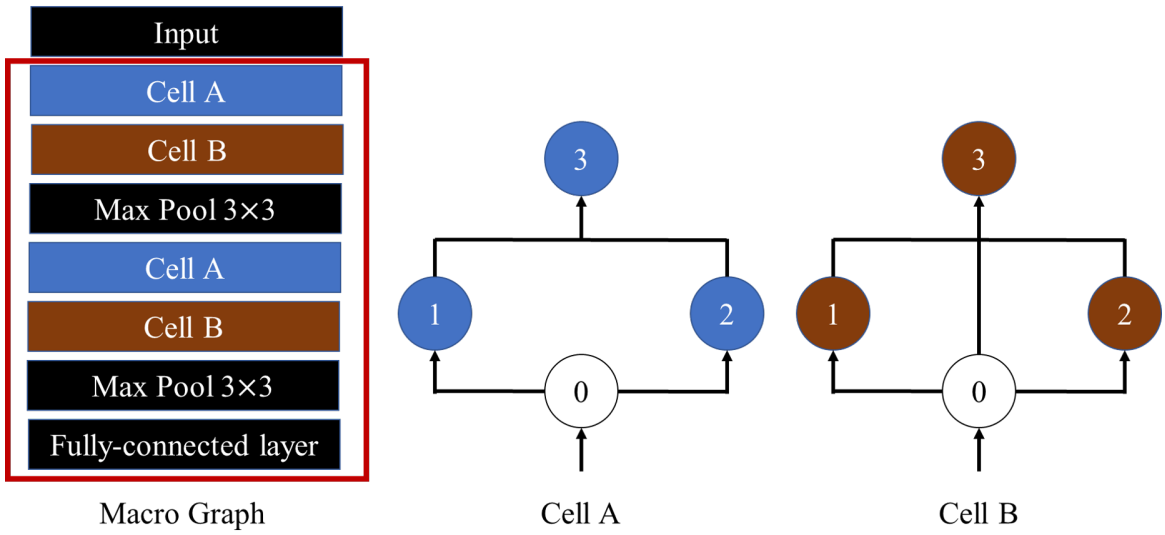

Figure 7: The example of macro graph and cells. Macro architecture is built by assembling cells and other components. Cell is composed of operations such as 3×3 convolution and connections.

10.1.3 Hierarchical representation

Hierarchical representation searches different levels of components and assembles these components as directed acyclic graph (DAG) in the end [21]. The level-1 primitives are the component such as 3×3 convolution and max pooling. The level $n+1$ motifs are the combinations of level n motifs. Since the hierarchical representation assembles the motifs, we **can eliminate conditional parameters** and **make them categorical parameters**. This search space also allows repeated patterns in the architecture and to reuse various levels of motifs. The major advantage over the cell search space is more flexibility and it **allows different blocks in different parts** of the network. On the other hand, since it is more flexible, the **search space is larger** than the cell search space; therefore, the optimization may potentially much harder.

10.2 Black-box optimization

Neural architecture can also be optimized by black-box optimization. The major difference from hyperparameter optimization is the existence of many conditional parameters and categorical parameters.

One of the earliest works is **neuroevolution**. Neuroevolution **evolves architecture jointly with its weights**. It first initializes a population with N random architectures. Then it repeats sampling with replacement from the population based on the fitness and while applying mutation to the N individuals. Note that mutations include adding, changing or removing a layer and we can choose elitism to preserve the elites. Another method is **Bayesian optimization** such as TPE, SMAC and NASBOT [15] [30]. They **jointly optimize architecture and hyperparameters**. **Regularized evolution** also shows great performance [26]. It promotes the exploration of architectures by dropping older architectures from the population and updates the weights by SGD. The regularized evolution encodes the architecture space by categorical hyperparameters. Additionally, the combination of **reinforcement learning (RL) and recurrent neural networks (RNN)** has been studied actively these days [37]. The RNN called **controller** generates a neural architecture (called **child net**) as a sequence [31] and evaluates the child net on a validation dataset. Then the weights of RNN is updated

[30] It uses arc-kernel [33] that is invented for NAS and can handle conditional parameters.

[31] Each state is the hidden states and each action is to take a component. RNN approximates the Q-value function taking a hidden state and a component as arguments.

using RL based on the reward that is the performance on the validation dataset.

10.3 Speed-up techniques

Since NAS requires substantial amount of time, it is essential to reduce computational time. The major methods are **multi-fidelity optimization**, **learning curve prediction** and **meta-learning** [10, 20] [32]. Other examples are **network morphisms** [7] and **weight sharing and one-shot models** [1, 25]. Network morphisms adds new operations $o(\boldsymbol{x}; \boldsymbol{w}_{\text{new}})$ on the existing architecture $\mathcal{A}(\boldsymbol{x}; \boldsymbol{w})$ so that the new architecture satisfies the following property:

$$\mathcal{A}(\boldsymbol{x}; \boldsymbol{w}) = o(\mathcal{A}(\boldsymbol{x}; \boldsymbol{w}); \boldsymbol{w}_{\text{new}}).$$

In other words, the $o(\boldsymbol{x}; \boldsymbol{w}_{\text{new}})$ is an identity function. By transferring the weights from the previous architecture and training jointly with the new operation $o(\cdot)$, we can save the large amount of training time. The drawback of this method is the growing size of the architecture [33]. Therefore, the multi-objective (the performance and the computational time) optimization is required for the network morphisms [8]. The last method is weight sharing and one-shot models. One-shot models first construct the architecture graph, called **supergraph**, with all the possible operations, that are **edges**, and the weights in between those operations that are called **nodes**. Each subgraph shares the same weights with the supergraph. Then we search the optimal subgraph among the supergraphs while training the supergraph. Since we need to train the supergraph only once, The total computational cost can be significantly reduced, Note that since the HPO problems do not handle architecture parameters, network morphisms and one-shot models are not used in HPO.

10.4 One-shot models

10.4.1 Differentiable architecture search (DARTS)

Since the choices of operation and conditional parameters are not differentiable, it is not possible to apply gradient methods directly. Therefore, the DARTS [22] introduces the following **stochastic relaxation**:

$$\boldsymbol{z}^{(j)} = \sum_{i<j} \tilde{o}^{(i,j)}(\boldsymbol{z}^{(i)}) = \sum_{i<j} \sum_{o \in \mathcal{O}^{(i,j)}} \frac{\exp(\alpha_o^{(i,j)})}{\sum_{o' \in \mathcal{O}^{(i,j)}} \exp(\alpha_{o'}^{(i,j)})} o(\boldsymbol{z}^{(i)})$$

where $\boldsymbol{z}^{(j)}$ is the feature vector after passing the j-th node weight tensor, $\mathcal{O}^{(i,j)}$ is the set of operations between the i- and j-th nodes and $\alpha_o^{(i,j)}$ is the weight of the operation o. Roughly speaking, DARTS uses soft-choice shown in Figure 8 of operations instead of hard-choice and that is why the architecture is differentiable as in the case of softmax loss. In the end, this method chooses the operation that has the largest $\alpha_o^{(i,j)}$ among the set $\mathcal{O}^{(i,j)}$ and discretizes the architecture. The major issues of DARTS are the **hyperparameter sensitivity** and **memory consumption**. The first issue is addressed by early-stopping based on the curvature [34] of the validation loss [36] or adversarial training to avoid sharp minima [3]. The memory issue is addressed by GDAS [5], which keeps only a single architecture sampled from a Gumbel softmax distribution in memory or PC-DARTS [35], which performs the search on a subset of the channels in the one-shot model. Another example is ProxyNAS [2].

10.4.2 Other one-shot models

Major examples are **DropPath** [1] and **sampling of sub-architectures** [18, 29]. One of the major issues of one-shot model is the co-adaptation of weights. In other words, each weight depends on each

[32] Since there are not many datasets available for DL, there are only little work on meta-learning for NAS.

[33] Since it is hard to yield an identity mapping by reducing the size, the network morphisms realize the mapping by adding components.

[34] The curvature is computed from eigenvalues of the Hessian matrix. Since DARTS discretizes in the end and the architecture may stay close to the sharp local minima, it can lead to degenerate architecture.

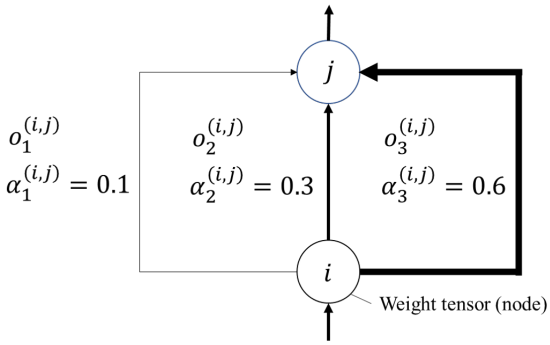

Figure 8: One-shot models set all the possible operations ($o_1^{(i,j)}, o_2^{(i,j)}, o_3^{(i,j)}$) as shown in the figure. Other NAS algorithms chooses only one of the operations (hard choice). However, DARTS chooses the operation by soft choice. In the implementation, the DARTS takes the weighted sum of the outputs from each operation during search and chooses the best operation in the end.

Algorithm 1 DARTS (Bi-level optimization)

function DARTS ▷ Each update is performed by SGD.
  while The budget is left **do**
    $\boldsymbol{w} = \boldsymbol{w} - \frac{\partial \mathcal{L}_{\text{train}}(\boldsymbol{w}, \boldsymbol{\alpha})}{\partial \boldsymbol{w}}$ ▷ Training (Inner loop)
    $\boldsymbol{\alpha} = \boldsymbol{\alpha} - \frac{\partial \mathcal{L}_{\text{valid}}(\boldsymbol{w}, \boldsymbol{\alpha})}{\partial \boldsymbol{\alpha}}$ ▷ Architecture search (Outer loop)
  return $\text{argmax}_{o \in \mathcal{O}^{(i,j)}} \alpha_o^{(i,j)}$ for each edge (i, j)

other. To avoid this issue, DropPath introduces the dropout of edges, i.e. to disable some architecture components, during training and ScheduledDropPath schedule the dropout rate for this. Another example of one-shot models is sampling of sub-architectures. This method picks only one architecture from the search space and updates the weight parameters based on the architecture. The most naïve method samples architectures from uniform distribution and **ENAS** [25] samples architectures from the learned policy of a RNN controller.

10.4.3 The usage of the trained one-shot model

After the training of an one-shot model, we have to choose architecture except DARTS. We can choose an individual as follows:

- Sample M architectures uniformly
- Rank them based on the validation error using the one-shot model weights
- Return the top performing architecture to train from scratch

If the performance of retrained models and that of models with weights from the one-shot model highly correlate, this selection strategy works well. Otherwise, it ends up with suboptimal architecture [1]. Note that the HPO for NAS is also requried and some papers suggest the joing optimization [27, 28].

11 Generative models

11.1 Auto-encoder

Auto-encoder is a generative model that reconstructs an input $\boldsymbol{x}$ by two neural networks. Auto-encoder first encodes an input $\boldsymbol{x}$ into latent representation $\boldsymbol{z} = f^e(\boldsymbol{x}; \boldsymbol{w}_e)$. This representation is

compressed lower dimensional representation of the input. Then it decodes the latent representation and reconstructs the original input $\hat{\boldsymbol{x}} = f^d(\boldsymbol{x}; \boldsymbol{w}_d)$. The neural networks learn the discrepancy between the original input and the reconstructed input. Therefore, it does not require labels and it is called **unsupervised learning**. The conceptual visualization is shown in Figure 9.

11.1.1 Regularized auto-encoders

Regularized auto-encoder has higher dimension of the latent space than the original input (called **over-complete representation**) and it puts a penalty on the magnitude of the variables by regularization. By the regularization, it throws out not important information and prevents pure copy. This model can choose the capacity based on the complexity of the data distribution.

11.1.2 Sparse auto-encoder

Sparse auto-encoder has the sparsity penalty as a regularization. The goal is to maximize the likelihood of the joint distribution of the reconstructed input and the latent vector. The joint distribution is formulated as follows:

$$p(\boldsymbol{x}, \boldsymbol{z}) = p(\boldsymbol{z})p(\boldsymbol{x}|\boldsymbol{z})$$

where $p(\boldsymbol{z})$ is a prior distribution and $p(\boldsymbol{x}|\boldsymbol{z})$ is a reconstructed distribution or likelihood given $\boldsymbol{z}$ [35]. We obtain the following log-likelihood by marginalization:

$$\log p(\boldsymbol{x}) = \log \int p(\boldsymbol{x}, \boldsymbol{z}) d\boldsymbol{z}.$$

In essence, the latent vector is determined by a parametric model, so we maximize the following:

$$\log p(\boldsymbol{x}, \boldsymbol{z}) = \log p(\boldsymbol{x}|\boldsymbol{z}) + \log p(\boldsymbol{z}).$$

We can achieve the sparsity by introducing Laplace prior $p(\boldsymbol{z}) = \frac{\lambda}{2} \exp(-\lambda|\boldsymbol{z}|)$. In general, the sparsity often increases generalization and avoids overfitting. This auto-encoder is often used as an unsupervised pre-processing step for downstream tasks. Note that downstream tasks are to transfer pre-trained weights to supervised learning tasks such as classification tasks.

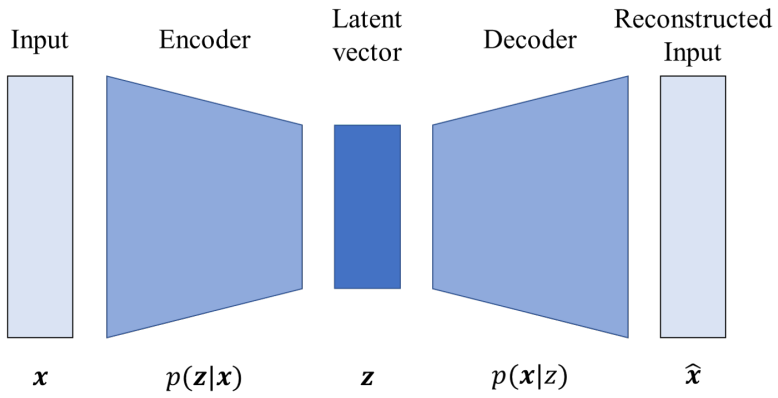

Figure 9: The conceptual visualization of auto-encoder. Both encoder and decoder are neural networks.

[35] Strictly speaking, these $\boldsymbol{x}$ is $\hat{\boldsymbol{x}}$. However, we unite the notation to $\boldsymbol{x}$ and consider the distribution as the likelihood.

11.1.3 Denoising auto-encoder

Denoising auto-encoder learns to cancel out the effect of noise corruption rather than adding an explicit regularization. It minimizes the loss function $L(\boldsymbol{x}, f^d(f^e(\tilde{\boldsymbol{x}})))$ where $\tilde{\boldsymbol{x}}$ is a corrupted input. To realize the corruption, we introduce a **corruption process** $C(\tilde{\boldsymbol{x}}|\boldsymbol{x})$ after the input. The optimization is performed by the gradient descent with respect to the following negative log-likelihood:

$$-\mathbb{E}_{\boldsymbol{x}\sim\mathcal{D},\tilde{\boldsymbol{x}}\sim C(\tilde{\boldsymbol{x}}|\boldsymbol{x})} \log p(\boldsymbol{x}|\boldsymbol{z} = f^e(\tilde{\boldsymbol{x}})).$$

As a result, we obtain the reconstructed distribution $p(\boldsymbol{x}|\tilde{\boldsymbol{x}})$. Note that the latent vector is deterministic in this method, but **if we make it stochastic, it is equivalent to variational auto-encoder**.

11.1.4 Variational auto-encoder (VAE)

In VAE, we assume that each data has important factors and VAE explicitly models such factors. In other words, $\boldsymbol{z}$ reflects the **causal factors** $p(\boldsymbol{x}|\boldsymbol{z})$ that makes it easier to reconstruct $p(\boldsymbol{x}) = \int p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})d\boldsymbol{z}$ [36]. To achieve it, we minimize the KL-divergence between the posterior and the parametric distribution rather than the optimization of the log-likelihood. Once we have a good approximation of $p(\boldsymbol{x})$, we can generate new data that are similar, but different from the original data.

To obtain such a model, we first need to infer the posterior distribution of the latent vector $p(\boldsymbol{z}|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})}{\int p(\boldsymbol{x},\boldsymbol{z})d\boldsymbol{z}}$. Since the integral in the denominator is intractable, we use **approximate inference**. The approximate inference uses a simpler parametric distribution $q(\boldsymbol{z}|\boldsymbol{x};\boldsymbol{w}_e)$ and minimizes the KL-divergence between $q(\boldsymbol{z}|\boldsymbol{x};\boldsymbol{w}_e)$ and $p(\boldsymbol{z}|\boldsymbol{x})$. The equation is formulated as follows:

$$\begin{aligned}
D_{\mathrm{KL}}(q(\boldsymbol{z}|\boldsymbol{x};\boldsymbol{w}_e)\|p(\boldsymbol{z}|\boldsymbol{x})) &= \int q(\boldsymbol{z}|\boldsymbol{x};\boldsymbol{w}_e) \log \frac{q(\boldsymbol{z}|\boldsymbol{x};\boldsymbol{w}_e)}{p(\boldsymbol{z}|\boldsymbol{x})} d\boldsymbol{z} \\
&= \int q(\boldsymbol{z}|\boldsymbol{x};\boldsymbol{w}_e) \left(\log q(\boldsymbol{z}|\boldsymbol{x};\boldsymbol{w}_e) - \log \frac{p(\boldsymbol{x},\boldsymbol{z})}{p(\boldsymbol{x})} \right) d\boldsymbol{z} \\
&= \underbrace{\int q(\boldsymbol{z}|\boldsymbol{x};\boldsymbol{w}_e) \left(\log q(\boldsymbol{z}|\boldsymbol{x};\boldsymbol{w}_e) - \log p(\boldsymbol{x},\boldsymbol{z}) \right) d\boldsymbol{z}}_{=-\mathcal{L}(\boldsymbol{x},q),\ (\log p(\boldsymbol{x}) \geq \mathcal{L}(\boldsymbol{x},q))} + \log p(\boldsymbol{x}).
\end{aligned}$$

Since $\mathcal{L}(\boldsymbol{x}, \boldsymbol{w}_e)$ is the lower bound of $\log p(\boldsymbol{x})$, it is called **evidence lower bound (ELBO)**. The goal is to minimize the KL-divergence and it is achieved when we maximize the ELBO. This is because $\log p(\boldsymbol{x})$ is a constant number and when we maximize the ELBO, KL divergence approaches zero. The ELBO is reformulated as follows:

$$\begin{aligned}
\mathcal{L}(\boldsymbol{w}_e, \boldsymbol{w}_d, \boldsymbol{x}) &= \int q(\boldsymbol{z}|\boldsymbol{x};\boldsymbol{w}_e) \left(\log p(\boldsymbol{x},\boldsymbol{z};\boldsymbol{w}_d) - \log q(\boldsymbol{z}|\boldsymbol{x};\boldsymbol{w}_e) \right) d\boldsymbol{z} \\
&= \int q(\boldsymbol{z}|\boldsymbol{x};\boldsymbol{w}_e) \left(\log p(\boldsymbol{x}|\boldsymbol{z};\boldsymbol{w}_d) + \log p(\boldsymbol{z}) - \log q(\boldsymbol{z}|\boldsymbol{x};\boldsymbol{w}_e) \right) d\boldsymbol{z} \\
&= \underbrace{-D_{\mathrm{KL}}(q(\boldsymbol{z}|\boldsymbol{x};\boldsymbol{w}_e)\|p(\boldsymbol{z}))}_{\text{regularization term}} + \underbrace{\int q(\boldsymbol{z}|\boldsymbol{x};\boldsymbol{w}_e) \log p(\boldsymbol{x}|\boldsymbol{z};\boldsymbol{w}_d) d\boldsymbol{z}}_{\text{reconstruction term}}
\end{aligned}$$

where $p(\boldsymbol{z})$ is a prior distribution. Note that the deterministic VAE with respect to the latent vector does not have the self-regularization term. The naïve implementation requires sampling to take the expectation. To handle the issue, the **reparametrization trick** has been invented. Reparametrization trick parametrizses the parameters of q. For example, if q is a gaussian distribution, the parameters are mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\sigma}$, so we parametrizse them as follows:

$$\boldsymbol{z} = \boldsymbol{\mu}(\boldsymbol{x}) + \boldsymbol{\sigma}(\boldsymbol{x}) \odot \boldsymbol{\epsilon}$$

[36] As mentioned previously, this is equivalent to the likelihood of the data distribution given the latent vector.

where ϵ is a random number sampled from $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$. In this case, the neural network for the encoder returns the mean vector and the covariance matrix. Since the forward pass is continuous and differentiable, we can apply the back propagation. The loss function is the following:

$$
\begin{aligned}
-\mathcal{L}(\boldsymbol{w}_e, \boldsymbol{w}_d, \boldsymbol{X}) &= \sum_{i=1}^{B} \left(D_{\mathrm{KL}}(q(\boldsymbol{z}|\boldsymbol{x}_i) \| p(\boldsymbol{z})) - \int q(\boldsymbol{z}|\boldsymbol{x}_i; \boldsymbol{w}_d) \log p(\boldsymbol{x}_i|\boldsymbol{z}; \boldsymbol{w}_d) d\boldsymbol{z} \right) \\
&= \sum_{i=1}^{B} \left(D_{\mathrm{KL}}(q(\boldsymbol{z}|\boldsymbol{x}_i) \| p(\boldsymbol{z})) - \sum_{d=1}^{D} \left(x_{i,d} \log \hat{x}_{i,d} + (1 - x_{i,d}) \log(1 - \hat{x}_{i,d}) \right) \right)
\end{aligned}
$$

where B is the batch size and D is the dimension of inputs. Intuitively speaking, the likelihood $p(\boldsymbol{x}_i|\boldsymbol{z}; \boldsymbol{w}_d)$ becomes smaller when the reconstruction error, i.e. the second term, becomes larger. It corresponds to the property of the negative log-likelihood. Note that the KL-divergence can be analytically computed in most cases of exponential family.

11.2 Generative Adversarial networks (GAN)

GAN is the model that jointly trains both generator and discriminator. The generator tries to fool the discriminator by the generated data and the discriminator tries to detects the real data as much as possible. When we define the output data by the generator as $f^g(\cdot; \boldsymbol{w}_g)$ and the likelihood of the real data as $f^d(\cdot; \boldsymbol{w}_d)$, the formulation is the following:

$$
\begin{aligned}
\textbf{Overall}: &\quad \min_{\boldsymbol{w}_g} \max_{\boldsymbol{w}_d} \left[\mathbb{E}_{\boldsymbol{x} \sim \mathcal{D}} \log f^d(\boldsymbol{x}; \boldsymbol{w}_d) + \mathbb{E}_{\boldsymbol{r} \sim p(\boldsymbol{r})} \log(1 - f^d(f^g(\boldsymbol{r}; \boldsymbol{w}_g); \boldsymbol{w}_d)) \right] \\
\textbf{Discriminator}: &\quad \max_{\boldsymbol{w}_d} \left[\mathbb{E}_{\boldsymbol{x} \sim \mathcal{D}} \log f^d(\boldsymbol{x}; \boldsymbol{w}_d) + \mathbb{E}_{\boldsymbol{r} \sim p(\boldsymbol{r})} \log(1 - f^d(f^g(\boldsymbol{r}; \boldsymbol{w}_g); \boldsymbol{w}_d)) \right] \\
\textbf{Generator}: &\quad \min_{\boldsymbol{w}_g} \left[\mathbb{E}_{\boldsymbol{r} \sim p(\boldsymbol{r})} \log(1 - f^d(f^g(\boldsymbol{r}; \boldsymbol{w}_g); \boldsymbol{w}_d)) \right]
\end{aligned}
$$

where $\boldsymbol{r}$ is a random noise vector and it is sampled each iteration. Note that the generative tasks are much more difficult that the classification tasks; therefore, we **train from the discriminator**. Since the discriminator requires the maximization, we apply gradient ascent for the update. The major issue of the naïve GAN is that the generator does not often work well in this setting. This is due to the small gradients when the generator does not work well, i.e. $f^d(f^g(\boldsymbol{r}; \boldsymbol{w}_g); \boldsymbol{w}_d)$ is close to 0. To alleviate the problem, we replace the original objective for the generator with the following:

$$
\max_{\boldsymbol{w}_g} \left[\mathbb{E}_{\boldsymbol{r} \sim p(\boldsymbol{r})} \log f^d(f^g(\boldsymbol{r}; \boldsymbol{w}_g); \boldsymbol{w}_d) \right].
$$

This objective promotes the learning when the generator works poorly.

12 Uncertainty handling

In some applications such as medical areas, more reliable decision making is required. In the same way, deep learning also requires uncertainty handling in some cases. Even though typical classification tasks use softmax and it can be interpreted probabilistically, it shows underconfidence for typically seen data and can show overconfidence for unrecognizable data. In other words, we would like to set the confidence higher for seen data and lower for unseen data. Therefore, we need additional uncertainty measures. There are two types of uncertainties: **Aleatoric uncertainty**, that is the intrinsic observation noise, and **Epistemic uncertainty**, that is the model uncertainty and can be reduced by more data. We discuss the measure of the combination of epistemic and aleatoric.

12.1 Properties of the Gaussian distribution

The Gaussian distribution has major four properties and we introduce them for the discussion later. Throuout this section, the Gaussian distribution takes the following form:

$$\begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{bmatrix}, \quad \begin{bmatrix} \boldsymbol{\Sigma}_{1,1} & \boldsymbol{\Sigma}_{1,2} \\ \boldsymbol{\Sigma}_{2,1} & \boldsymbol{\Sigma}_{2,2} \end{bmatrix} \right). \tag{7}$$

where $\boldsymbol{\mu}_i \in \mathbb{R}^{D_i}, \boldsymbol{\Sigma}_{i,j} \in \mathbb{R}^{D_i \times D_j}$ for $i, j = 1, 2$. The first property is realized by the marginalization $p(\boldsymbol{x}_1) = \int p(\boldsymbol{x}_1, \boldsymbol{x}_2) d\boldsymbol{x}_2$ and it is called **closed under marginalization** [37].

Theorem 4
Let $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ be jointly Gaussian distributed:

$$\begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{bmatrix}, \quad \begin{bmatrix} \boldsymbol{\Sigma}_{1,1} & \boldsymbol{\Sigma}_{1,2} \\ \boldsymbol{\Sigma}_{2,1} & \boldsymbol{\Sigma}_{2,2} \end{bmatrix} \right). \tag{8}$$

Then $\boldsymbol{x}_1 \sim \mathcal{N}(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_{1,1})$.

The second property is **closed under conditioning** and it is realized by $p(\boldsymbol{x}_2|\boldsymbol{x}_1) = \frac{p(\boldsymbol{x}_1, \boldsymbol{x}_2)}{p(\boldsymbol{x}_1)}$.

Theorem 5
Let $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ be jointly Gaussian distributed:

$$\begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{bmatrix}, \quad \begin{bmatrix} \boldsymbol{\Sigma}_{1,1} & \boldsymbol{\Sigma}_{1,2} \\ \boldsymbol{\Sigma}_{2,1} & \boldsymbol{\Sigma}_{2,2} \end{bmatrix} \right). \tag{9}$$

Then $\boldsymbol{x}_2|\boldsymbol{x}_1 \sim \mathcal{N}(\boldsymbol{\mu}_2 + \boldsymbol{\Sigma}_{1,2}^\top \boldsymbol{\Sigma}_{1,1}^{-1}(\boldsymbol{x}_1 - \boldsymbol{\mu}_1), \boldsymbol{\Sigma}_{2,2} - \boldsymbol{\Sigma}_{1,2}^\top \boldsymbol{\Sigma}_{1,1}^{-1} \boldsymbol{\Sigma}_{1,2})$.

Other properties are **closed under multiplication** and **linear maps**. Note that the Gaussian distribution is often used due to the convenient mathematical properties, but not because sampled data follow the distribution.

12.2 Bayesian linear regression

Bayesian linear regression gives an uncertainty measure to the linear regression. The linear regression assumes that the output $\boldsymbol{y}$ follows the Gaussian distribution with a fixed variance σ and uses **maximum likelihood estimation** on $p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w})$. On the other hand, Bayesian linear regression computes the posterior distribution of the weights $p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y})$ using Bayes' theorem as follows:

$$\begin{aligned} \text{posterior} &= \frac{\text{likelihood} \times \text{prior}}{\text{marginal likelihood}}, \\ p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}) &= \frac{p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}) p(\boldsymbol{w})}{\int p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}) p(\boldsymbol{w}) d\boldsymbol{w}}. \end{aligned}$$

Since the denominator $p(\boldsymbol{y}|\boldsymbol{X})$ does not depend on the weight $\boldsymbol{w}$ and the denominator generally requires a complicated integral, we use the maximum a posteriori (**MAP**) estimation [38]. In the MAP estimation, we maximize the following:

$$\begin{aligned} \text{posterior} &\propto \text{likelihood} \times \text{prior}, \\ p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}) &\propto p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}) p(\boldsymbol{w}). \end{aligned} \tag{10}$$

[37] It means we yield Gaussian distribution even after applying the operation.
[38] The maximum likelihood estimation is equivalent to the MAP estimation with the marginal likelihood that follows uniform distribution.

In this case, if we choose a prior that is conjugate to the likelihood, the posterior can often be analytically obtained. In the case of the likelihood following Gaussian distribution, the **conjugate prior** is Gaussian distribution. In other words, the following holds:

$$\begin{aligned} \text{prior}: \ & \boldsymbol{w} \sim \mathcal{N}(\boldsymbol{0}, \Sigma_{\text{prior}}), \\ \text{posterior}: \ & p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}) \sim \mathcal{N}(\boldsymbol{\mu}_{\text{post}}, \Sigma_{\text{post}}) \end{aligned}$$

where Σ_{prior} is a covariance matrix for the prior distribution, σ is a pre-defined noise, $\boldsymbol{\mu}_{\text{post}} = \frac{1}{\sigma^2}\Sigma_{\text{post}}\boldsymbol{X}^\top\boldsymbol{y}$ and $\Sigma_{\text{post}} = \left(\frac{1}{\sigma^2}\boldsymbol{X}^\top\boldsymbol{X} + \Sigma_{\text{prior}}^{-1}\right)^{-1}$. These values can be analytically derived using log function on Eq. (10). Note that since Σ_{prior} controls the regularization effect and affects the performance, the choice of Σ_{prior} is called **model selection** and it is usually performed by cross validation. Using the weights sampled from the posterior, the prediction of Bayesian linear regression is computed as follows:

$$\begin{aligned} p(\hat{y}|\boldsymbol{x}, \boldsymbol{X}, \boldsymbol{y}) &= \int p(\hat{y}|\boldsymbol{x}, \boldsymbol{w}) p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}) d\boldsymbol{w} \\ &= \mathcal{N}(\boldsymbol{x}^\top\boldsymbol{\mu}_{\text{post}}, \boldsymbol{x}^\top\Sigma_{\text{post}}\boldsymbol{x}). \end{aligned}$$

Note that we can realize the Bayesian non-linear regression by replacing the $\boldsymbol{x}$ with non-linearly mapped elements such as $\Phi = [1, x, x^2, ..., x^D]$ as long as **the computation is linear in $\boldsymbol{w}$**. Furthermore, although the Laplace prior is not conjugate to the Gaussian likelihood and it requires to solve non-linear system, we can introduce **the Laplace prior to promote the sparsity** of the model.

12.3 Neural networks with Bayesian

We discuss how we can use the linear regression in a Bayesian manner. We will expand the discussion and consider how we use the Bayesian concept for neural networks.

12.3.1 DNGO

DNGO [32] first trains a neural network on a given dataset and use the feature vector obtained in the last hidden layer as basis functions of the Bayesian linear regression.

12.3.2 Bayesian neural networks

Bayesian neural networks learn the posterior of weights $p(\boldsymbol{w}|\mathcal{D})$ and predict by the follwoing marginalization:

$$p(y|\boldsymbol{x}, \mathcal{D}) = \int p(y|\boldsymbol{x}, \boldsymbol{w}) p(\boldsymbol{w}|\mathcal{D}) d\boldsymbol{w}.$$

The core part of Bayesian neural networks are to compute the posterior:

$$p(\boldsymbol{w}|\mathcal{D}) = p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}) = \frac{p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}) p(\boldsymbol{w})}{\int p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}) p(\boldsymbol{w}) d\boldsymbol{w}}.$$

Since the Gaussian distribution is not closed under non-linear mapping and neural networks include non-linear mapping, the likelihood $p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w})$ is complicated and the posterior approximation requires Markov-chain Monte-carlo method or variational inference. In the next section, we will discuss both methods.

12.4 Posterior inference for non-linear models

12.4.1 Variational inference

As in VAE, the variational inference minimizes the KL-divergence between the posterior $p(\boldsymbol{w}|\mathcal{D})$ and a simple paraemteric model $q(\boldsymbol{w};\theta)$ as follows:

$$\begin{aligned}
D_{\mathrm{KL}}(q(\boldsymbol{w};\theta)\|p(\boldsymbol{w}|\mathcal{D}))) &= \int q(\boldsymbol{w};\theta)\log\frac{q(\boldsymbol{w};\theta)}{p(\boldsymbol{w}|\mathcal{D})}d\boldsymbol{w} \\
&= \int q(\boldsymbol{w};\theta)\Bigg(\log q(\boldsymbol{w};\theta) - \log\frac{p(\mathcal{D}|\boldsymbol{w})p(\boldsymbol{w})}{p(\mathcal{D})}\Bigg)d\boldsymbol{w} \\
&= \int q(\boldsymbol{w};\theta)\Big(\log q(\boldsymbol{w};\theta) - \log p(\mathcal{D}|\boldsymbol{w}) - \log p(\boldsymbol{w})\Big)d\boldsymbol{w} + \log p(\mathcal{D}) \\
&= D_{\mathrm{KL}}(q(\boldsymbol{w};\theta)\|p(\boldsymbol{w})) - \underbrace{\mathbb{E}_{\boldsymbol{w}\sim q(\boldsymbol{w};\theta)}[\log p(\mathcal{D}|\boldsymbol{w})]}_{\text{log-likelihood}} + \underbrace{\log p(\mathcal{D})}_{\text{constant}} \\
&= -\underbrace{(\mathbb{E}_{\boldsymbol{w}\sim q(\boldsymbol{w};\theta)}[\log p(\mathcal{D}|\boldsymbol{w})] - D_{\mathrm{KL}}(q(\boldsymbol{w};\theta)\|p(\boldsymbol{w})))}_{\text{ELBO: }\mathcal{L}(\theta)} + \text{const.}
\end{aligned}$$

Since the lower bound of KL-divergence is zero and the sum of the KL-divergence and the ELBO is constant, the minimization of the KL-divergence is realized by the maximization of the ELBO $\mathcal{L}(\theta)$. Furthermore, the KL-divergence term works as regularization, because we have to make the distribution closer to the prior. In contrast to MCMC, the variational inference avoids sampling by taking a parametric model and approximates the model by the maximization of ELBO. Note that the implementation requires the computation over the whole dataset and the expecation requires Monte-carlo sampling.

12.4.2 Marcov-chain Monte-carlo method (MCMC)

Another posterior estimation is performed by MCMC. MCMC samples each weights according to a **proposal distribution** or **transition distribution** $p(\boldsymbol{w}_{t+1}|\boldsymbol{w}_t)$ and moves around the space while accepting or rejecting the proposal from the distribution. Since the next state is determined only by the current state, it is called Markov-chain. For the final sampling, we use every t-th sample from the history to avoid the correlation between samples close to each other in terms of time steps. The goal of the sampling is to identify the **stationary distribution** π that is ideally the posterior distribution $p(\boldsymbol{w}|\mathcal{D})$. The sufficient condition for a stationary distribution to exist is that the **detailed balance** $\pi(\boldsymbol{w}_t)p(\boldsymbol{w}_{t+1}|\boldsymbol{w}_t) = \pi(\boldsymbol{w}_{t+1})p(\boldsymbol{w}_t|\boldsymbol{w}_{t+1})$. In practice, even when the detailed balance is satisfied, it may still take time to reach the stationary distribution. This time (from the beginning) is called **mixing time**. Intuitively, the distribution reaches the stationary distribution when the chain forgets the beginning states. Therefore, we take samples after a burning-in phase and the length of the burning-in phase is a hyperparameter. MCMC can sample from multi-modal distributions, but it often fails if each peak is far away from each other.

The major algorithm for MCMC is Metropolis-Hasting shown in Algorithm 2. Since it takes the ratio of the distribution, we do not have to explicitly compute the constant of proportionality $p(\mathcal{D})$. Therefore, we can transform the posterior as follows:

$$p(\boldsymbol{w}|\mathcal{D}) \propto p(\mathcal{D}|\boldsymbol{w})p(\boldsymbol{w}) \text{ (likelihood} \times \text{prior)}.$$

We use the right hand side of this equation for the computation of the acceptance rate. Another example is Hamiltonian Monte Carlo (HMC) that takes into account gradients of weights. The special case is the following Langevin dynamics:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \frac{\alpha}{2}\Bigg(\underbrace{\frac{\partial\log p(\boldsymbol{w}_t)}{\partial\boldsymbol{w}}}_{\text{prior gradient}} + \underbrace{\frac{\partial\log p(\mathcal{D}|\boldsymbol{w}_t)}{\partial\boldsymbol{w}}}_{\text{likelihood gradient}}\Bigg) + \boldsymbol{r}_t$$

Algorithm 2 Metropolis Hastings

$p(\boldsymbol{w}'|\boldsymbol{w})$ ▷ Proposal distribution. This is typically Gaussian distribution with mean = $\boldsymbol{w}$.
$\pi(\boldsymbol{w}) = p(\boldsymbol{w}|\mathcal{D}) \propto p(\mathcal{D}|\boldsymbol{w})p(\boldsymbol{w})$ ▷ Since we take the ratio, $p(\mathcal{D})$ is canceled out.

1: **function** Metropolis Hastings
2: **for** $t = 0, 1, \ldots, T$ **do**
3: $\boldsymbol{w}' \sim p(\boldsymbol{w}'|\boldsymbol{w}_t)$
4: $\boldsymbol{w}_{t+1} = \boldsymbol{w}'$ with the probability of $\min\left(1, \frac{\pi(\boldsymbol{w}')p(\boldsymbol{w}_t|\boldsymbol{w}')}{\pi(\boldsymbol{w}_t)p(\boldsymbol{w}'|\boldsymbol{w}_t)}\right)$ otherwise $\boldsymbol{w}_t$

where $\boldsymbol{r}_t$ are random numbers that follow Gaussian distribution and α is a learning rate. The major issue of this gradient is to take the whole dataset to compute the likelihood gradient. To ease this issue, the stochastic gradient Langevin dynamics (SGLD) has been invented. Another solution is to use mini-batches and this is SGHMC.

12.5 Alternative uncertainty treatments for neural networks

12.5.1 Output ensembling

A naïve way to measure the uncertainty of regression tasks is the following:

$$\hat{y}(\boldsymbol{x}) \sim \mathcal{N}(\mu(\boldsymbol{x}), \sigma(\boldsymbol{x})^2)$$
$$\text{where } \mu(\boldsymbol{x}) = \frac{1}{M}\sum_{i=1}^{M} f_i(\boldsymbol{x}; \boldsymbol{w}_i),\ \sigma^2(\boldsymbol{x}) = \frac{1}{M}\sum_{i=1}^{M}(f_i(\boldsymbol{x}; \boldsymbol{w}_i) - \mu)^2$$

and $f_i(\cdot; \boldsymbol{w}_i)$ is the i-th neural network and we take only the average, but not the variance, of softmax in the case of classification tasks. Depending on how we collect the M neural networks, the performance and the computational overhead will change. The simple methods are to collect models by **randomly dropping weights (MC dropout)** of a trained model or collect M **different weights from MCMC**. The **snapshot ensemble**, that uses a cyclic learning rate scheduing and collects weights at each end of the cycles, is another option. Those three methods **require only 1 training time**. Another effective method is to train different M models by different initial weights using SGD. It requires M training, but it achieves the state-of-the-art performance. Other than that, models with different hyperparameter configurations can take advantage of hyperparameter optimization and M different architectures can diversify the neural architectures and outperform the naïve ensembles.

12.5.2 The estimation of parametric models

The most common model is the Gaussian distribution and we estimate the parameters μ, σ. The performance metric of the training is the loglikelihood as follows:

$$\log p(\mathcal{D}|\boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N} \log p(y_i|\mu(\boldsymbol{x}_i; \boldsymbol{w}), \sigma^2(\boldsymbol{x}_i; \boldsymbol{w}))$$

where $p(y|\mu, \sigma^2)$ is the probability density value of y in the Gaussian distribution with the mean μ and the variance σ^2. We can combine the parametric model and ensembling as follows:

$$\mu = \frac{1}{M}\sum_{i=1}^{M}\mu_i,\ \sigma^2 = \frac{1}{M}\sum_{i=1}^{M}((\mu_i - \mu)^2 + \sigma_i^2).$$

Note that this variance is the total variance.

References

[1] Bender, G., Kindermans, P.J., Zoph, B., Vasudevan, V., Le, Q.: Understanding and simplifying one-shot architecture search. In: International Conference on Machine Learning. pp. 550–559. PMLR (2018)

[2] Cai, H., Zhu, L., Han, S.: Proxylessnas: Direct neural architecture search on target task and hardware. arXiv preprint arXiv:1812.00332 (2018)

[3] Chen, X., Hsieh, C.J.: Stabilizing differentiable architecture search via perturbation-based regularization. In: International Conference on Machine Learning. pp. 1554–1565. PMLR (2020)

[4] Domhan, T., Springenberg, J.T., Hutter, F.: Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In: Twenty-fourth international joint conference on artificial intelligence (2015)

[5] Dong, X., Yang, Y.: Searching for a robust neural architecture in four gpu hours. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 1761–1770 (2019)

[6] Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. Journal of machine learning research **12**(7) (2011)

[7] Elsken, T., Metzen, J.H., Hutter, F.: Simple and efficient architecture search for cnns. In: Workshop on Meta-Learning at NIPS (2017)

[8] Elsken, T., Metzen, J.H., Hutter, F.: Efficient multi-objective neural architecture search via lamarckian evolution. arXiv preprint arXiv:1804.09081 (2018)

[9] Elsken, T., Metzen, J.H., Hutter, F.: Neural architecture search: A survey. The Journal of Machine Learning Research **20**(1), 1997–2017 (2019)

[10] Elsken, T., Staffler, B., Metzen, J.H., Hutter, F.: Meta-learning of neural architectures for few-shot learning. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 12365–12375 (2020)

[11] Feurer, M., Springenberg, J., Hutter, F.: Initializing bayesian hyperparameter optimization via meta-learning. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 29 (2015)

[12] Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572 (2014)

[13] Hanin, B., Rolnick, D.: Complexity of linear regions in deep networks. In: International Conference on Machine Learning. pp. 2596–2604. PMLR (2019)

[14] Kandasamy, K., Dasarathy, G., Schneider, J., Póczos, B.: Multi-fidelity bayesian optimisation with continuous approximations. In: International Conference on Machine Learning. pp. 1799–1808. PMLR (2017)

[15] Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., Xing, E.: Neural architecture search with bayesian optimisation and optimal transport. arXiv preprint arXiv:1802.07191 (2018)

[16] Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)

[17] Klein, A., Falkner, S., Bartels, S., Hennig, P., Hutter, F.: Fast bayesian optimization of machine learning hyperparameters on large datasets. In: Artificial Intelligence and Statistics. pp. 528–536. PMLR (2017)

[18] Li, L., Talwalkar, A.: Random search and reproducibility for neural architecture search. In: Uncertainty in artificial intelligence. pp. 367–377. PMLR (2020)

[19] Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., Talwalkar, A.: Hyperband: A novel bandit-based approach to hyperparameter optimization. The Journal of Machine Learning Research **18**(1), 6765–6816 (2017)

[20] Lian, D., Zheng, Y., Xu, Y., Lu, Y., Lin, L., Zhao, P., Huang, J., Gao, S.: Towards fast adaptation of neural architectures with meta learning. In: International Conference on Learning Representations (2019)

[21] Liu, H., Simonyan, K., Vinyals, O., Fernando, C., Kavukcuoglu, K.: Hierarchical representations for efficient architecture search. arXiv preprint arXiv:1711.00436 (2017)

[22] Liu, H., Simonyan, K., Yang, Y.: Darts: Differentiable architecture search. arXiv preprint arXiv:1806.09055 (2018)

[23] Loshchilov, I., Hutter, F.: Decoupled weight decay regularization. arXiv preprint arXiv:1711.05101 (2017)

[24] Maclaurin, D., Duvenaud, D., Adams, R.: Gradient-based hyperparameter optimization through reversible learning. In: International conference on machine learning. pp. 2113–2122. PMLR (2015)

[25] Pham, H., Guan, M., Zoph, B., Le, Q., Dean, J.: Efficient neural architecture search via parameters sharing. In: International Conference on Machine Learning. pp. 4095–4104. PMLR (2018)

[26] Real, E., Aggarwal, A., Huang, Y., Le, Q.V.: Regularized evolution for image classifier architecture search. In: Proceedings of the aaai conference on artificial intelligence. vol. 33, pp. 4780–4789 (2019)

[27] Runge, F., Stoll, D., Falkner, S., Hutter, F.: Learning to design rna. arXiv preprint arXiv:1812.11951 (2018)

[28] Saikia, T., Marrakchi, Y., Zela, A., Hutter, F., Brox, T.: Autodispnet: Improving disparity estimation with automl. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 1812–1823 (2019)

[29] Saxena, S., Verbeek, J.: Convolutional neural fabrics. Advances in neural information processing systems **29**, 4053–4061 (2016)

[30] Sen, R., Kandasamy, K., Shakkottai, S.: Multi-fidelity black-box optimization with hierarchical partitions. In: International conference on machine learning. pp. 4538–4547. PMLR (2018)

[31] Smith, S.L., Kindermans, P.J., Ying, C., Le, Q.V.: Don't decay the learning rate, increase the batch size. arXiv preprint arXiv:1711.00489 (2017)

[32] Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., Patwary, M., Prabhat, M., Adams, R.: Scalable bayesian optimization using deep neural networks. In: International conference on machine learning. pp. 2171–2180. PMLR (2015)

[33] Swersky, K., Duvenaud, D., Snoek, J., Hutter, F., Osborne, M.A.: Raiders of the lost architecture: Kernels for bayesian optimization in conditional parameter spaces. arXiv preprint arXiv:1409.4011 (2014)

[34] Swersky, K., Snoek, J., Adams, R.P.: Multi-task bayesian optimization (2013)

[35] Xu, Y., Xie, L., Zhang, X., Chen, X., Qi, G.J., Tian, Q., Xiong, H.: Pc-darts: Partial channel connections for memory-efficient architecture search. arXiv preprint arXiv:1907.05737 (2019)

[36] Zela, A., Elsken, T., Saikia, T., Marrakchi, Y., Brox, T., Hutter, F.: Understanding and robustifying differentiable architecture search. arXiv preprint arXiv:1909.09656 (2019)

[37] Zoph, B., Le, Q.V.: Neural architecture search with reinforcement learning. arXiv preprint arXiv:1611.01578 (2016)

[38] Zoph, B., Vasudevan, V., Shlens, J., Le, Q.V.: Learning transferable architectures for scalable image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 8697–8710 (2018)