Fundamental Framework of Machine Learning

Sunmook Choi felixchoi@korea.ac.kr

August 24, 2023

1 Risk Minimization

1.1 Empirical Risk Minimization

The principle of empirical risk minimization describes a mathematical formulation of supervised learning process. Before the formulation, we begin with a formal definition of the loss function [1] incurred by a function f at location x, given by an observation y.

Definition 1 (Loss Function)**.** *Denote by $(x, y, h(x)) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{Y}$ the triplet consisting of a feature vector x, an observation y and a prediction $h(x)$. The map $\ell \colon \mathcal{X} \times \mathcal{Y} \times \mathcal{Y} \to [0, \infty)$ with the property $\ell(x, y, y) = 0$ for all $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ will be called a loss function.*

Given a loss function, the problem of risk minimization, also known as supervised learning, is illustrated as below [2].

Definition 2 (Supervised Learning)**.** *Assume that there is some **fixed but unknown** joint distribution $P(X, Y)$ on $\mathcal{X} \times \mathcal{Y}$ where $X \in \mathcal{X}$ is a random (feature) vector and $Y \in \mathcal{Y}$ is a random target vector. The supervised learning process is to find a hypothesis (or a model) $h \in \mathcal{H}$ that describes the relationship between X and Y for some hypothesis class $\mathcal{H}$, i.e.,*

$$h^* = \arg\min_h R(h) = \int \ell(x, y, h(x)) P(x, y)\, dxdy, \tag{1.1}$$

where ℓ is a given loss function.

Here, we call $R(h)$ *the expected risk*. Unfortunately, the distribution P is unknown in the most practical cases, so that the integration is intractable. Instead of having access to P, we only have a set of observations, called training data, $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$, where

$(x_i, y_i) \sim P$ for all $i = 1, \ldots, n$. Using this training data $\mathcal{D}$, we may approximate P by *the empirical density*

$$P_{emp}(x, y) = \frac{1}{n} \sum_{i=1}^{n} \delta_{(x_i, y_i)}(x, y), \tag{1.2}$$

where $\delta_{(x_i, y_i)}$ denotes the δ-distribution, satisfying $\int \delta_{(x', y')}(x, y)\, f(x, y)\, dxdy = f(x', y')$. Using this empirical density, we now approximate expected risk by *the empirical risk*:

$$R_\delta(h) = \int \ell(x, y, h(x))\, P_{emp}(x, y)\, dxdy = \frac{1}{n} \sum_{i=1}^{n} \ell(x_i, y_i, h(x_i)). \tag{1.3}$$

In supervised learning, we find (or learn) $h \in \mathcal{H}$ by minimizing the empirical risk, and we call this process the *empirical risk minimization* (ERM).

Empirical risk minimization (ERM) is an efficient way to approximately minimize expected risk. In most of machine learning setups, after choosing a hypothesis class $\mathcal{H}$, an optimization algorithm is required to solve the problem:

$$h^* = \arg\min_{h \in \mathcal{H}} R_\delta(h) \tag{1.4}$$

1.1.1 Underfitting and Overfitting

Let's consider about the choice of a hypothesis class $\mathcal{H}$. If the class $\mathcal{H}$ has too small capacity, any model in this class cannot reduce the empirical risk sufficiently so that the model would perform terribly for unseen test samples. On the other hand, if the class $\mathcal{H}$ has too large capacity, then there should be a model $h \in \mathcal{H}$ such that it can 'memorize' all the data samples so that the empirical risk becomes zero. That is, ERM will find a model that memorizes the training samples rather than generalizes the training samples. Hence this may lead to the undesirable behavior of h outside the training data, which gives rise to great loss on test dataset. The former case is called *underfitting* while the latter is called *overfitting*.

When it comes to deep learning, the model class would be a subset of 'deep' neural networks that contains at least thousands of parameters. Therefore, overfitting problem is more common in deep learning practices rather than underfitting. The following subsections will describe how to handle the overfitting problems.

1.2 Structural Risk Minimization

The paper [2] illustrates some theoretical parts about ERM and introduces the principle of structural risk minimization (SRM) with regard to VC-dimensions. In this section, one example of SRM is described, which is widely used in the deep learning contexts.

Consider a hypothesis class $\mathcal{H} = \{h(x, w) \colon w \in W\}$, which is a set of neural networks. The parameters $\{w\}$ are the weights of the neural network. A structure is introduced through $\mathcal{H}_p = \{h(x, w) \colon \|w\| \leq C_p\}$ and $C_1 < C_2 < \cdots < C_n$. For a convex loss function, the minimization of the empirical risk within this element $\mathcal{H}_p$ of the structure is achieved through the minimization of

$$R_s(h, \gamma_p) = \frac{1}{n} \sum_{i=1}^{n} \ell(x_i, y_i, h(x_i, w)) + \gamma_p \|w\|^2 \tag{1.5}$$

with appropriately chosen Lagrange multipliers $\gamma_1 > \gamma_2 > \cdots > \gamma_n$. The well-known "weight decay" procedure refers to the minimization of this functional.

1.3 Vicinal Risk Minimization

Due to the undesired behavior of ERM, especially overfitting, it is required to improve empirical density P_{emp}. One way is to replace the delta function $\delta_{(x_i,y_i)}(x, y)$ or $\delta(x = x_i, y = y_i)$ that makes ERM constrained only on the training samples (so the model fails to generalize the data).

Vicinal risk minimization (VRM) [3] replaces the delta function into some probability distribution $\nu_{(x_i,y_i)}(\tilde{x}, \tilde{y})$ or $\nu(\tilde{x}, \tilde{y} \,|\, x_i, y_i)$ which describes the vicinity of the point (x_i, y_i). This is a way to consider not only samples that are observed but also the virtual feature-target pairs $(\tilde{x}, \tilde{y})$ of given samples (x_i, y_i). The formalization of VRM is described below.

Definition 3 (Vicinal Risk Minimization (VRM))**.** *Given a vicinity distribution ν, the true distribution P (in the ERM) is estimated by*

$$P_\nu(\tilde{x}, \tilde{y}) = \frac{1}{n} \sum_{i=1}^{n} \nu(\tilde{x}, \tilde{y} \,|\, x_i, y_i). \tag{1.6}$$

Then the vicinal risk is defined to be

$$R_\nu(h) = \frac{1}{m} \sum_{i=1}^{m} \ell(\tilde{x}_i, \tilde{y}_i, h(\tilde{x}_i)) \tag{1.7}$$

and the goal of VRM is to find $h \in \mathcal{H}$ that minimizes the vicinal risk, that is,

$$h^* = \arg\min_{h \in \mathcal{H}} R_\nu(h). \tag{1.8}$$

Vicinal risk minimization is also known as 'data augmentation'. Let's take a natural example of vicinity distribution, Gaussian vicinities:

$$\nu(\tilde{x}, \tilde{y} \,|\, x_i, y_i) = \mathcal{N}(\tilde{x} \,|\, x_i, \sigma^2) \cdot \delta(\tilde{y} = y_i). \tag{1.9}$$

This Gaussian vicinity distribution treats the data $\tilde{x}$ in a neighborhood of x_i to have the same target y_i as x_i. Here, the parameter σ controls the scale of density estimate. Note that the extreme case $\sigma = 0$ leads to ERM. More examples can be found in the paper [3].

1.3.1 mixup

We describe an example of VRM, called mixup [4]. Mixup is a data augmentation method formalized as a VRM with the following vicinal distribution:

$$\mu(\tilde{x}, \tilde{y} \,|\, x_i, y_i) = \frac{1}{n} \sum_{j}^{n} \mathbb{E}_\lambda \big[\delta(\tilde{x} = \lambda x_i + (1-\lambda) x_j,\ \tilde{y} = \lambda y_i + (1-\lambda) y_j)\big] \tag{1.10}$$

where $\lambda \sim Beta(\alpha, \alpha)$ for $\alpha \in (0, \infty)$. The mixup parameter α controls the strength of linear interpolation between feature-target pairs, recovering the ERM principle as $\alpha \to 0$. In a nutshell, sampling from the mixup vicinal distribution produces virtual feature-target vectors $(\tilde{x}, \tilde{y})$ such that

$$\tilde{x} = \lambda x_i + (1-\lambda) x_j,$$
$$\tilde{y} = \lambda y_i + (1-\lambda) y_j,$$

where $\lambda \in [0, 1]$, and where (x_i, y_i) and (x_j, y_j) are two feature-target vectors drawn at random from the training data. The paper also mentions alternative design choices of mixup and their performance.

1. Convex combinations of three or more examples with weights sampled from a Dirichlet distribution (the generalization of Beta distribution to multiple variables) does not provide further gain, but increases the computation cost of mixup.

2. Two feature-target pairs can be obtained from two distinct data loaders and then mixup is applied. Another way uses a single data loader to obtain one minibatch, and then mixup is applied to the same minibatch after random shuffling. Both strategies worked equally well, while the latter reduces the input/output requirements.

3. Interpolating only between inputs with equal label did not lead to the performance gains of mixup.

Then, what is the role of mixup? The mixup augmentation method encourages the model h to behave linearly in-between training examples. Authors of the paper argue that this linear behavior reduces the amount of undesirable oscillations for unseen test samples.

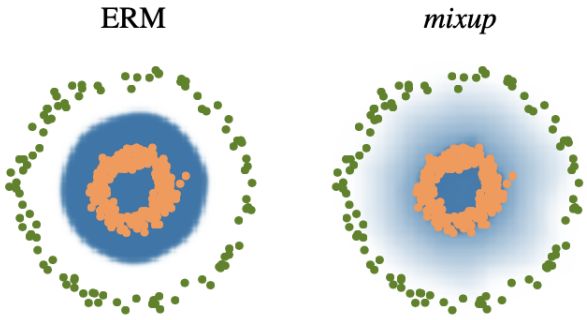
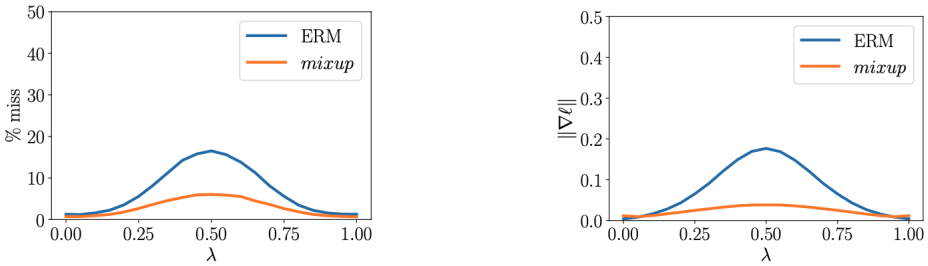


Figure 1.1: Effect of mixup ($\alpha = 1$) on a toy problem. Green: Class 0. Orange: Class 1. Blue shading indicates $p(y = 1|x)$.

Figure 1.3.1 shows that mixup leads to decision boundaries that transition linearly from class to class, providing a smoother estimate of uncertainty. Also, Figure 1.3.1 below illustrates the average behaviors of two models trained on the CIFAR-10 dataset using ERM and mixup. More fascinating experimental results are described in the paper [4].



(a) Prediction errors in-between training data. Evaluated at $x = \lambda x_i + (1-\lambda)x_j$, a prediction is counted as a "miss" if it does not belong to $\{y_i, y_j\}$. The model trained with *mixup* has fewer misses.

(b) Norm of the gradients of the model w.r.t. input in-between training data, evaluated at $x = \lambda x_i + (1-\lambda)x_j$. The model trained with *mixup* has smaller gradient norms.

Figure 1.2: mixup leads to more robust model behaviors in-between the training data.

2 Perceptron

The Perceptron is an artificial neural networks for binary classifiers. The algorithm was invented in 1958 by Frank Rosenblatt. Although the perceptron initially seemed promising, it was quickly proved that perceptrons could not be trained to recognize many kinds of patterns.

2.1 Assumptions of Perceptron

Definition 4 (Hyperplane)**.** *Let all data points lie on the d-dimensional space $\mathbb{R}^d$ with standard inner product. Here, we call $\mathbb{R}^d$ an ambient space. A hyperplane is a subspace of dimension $d-1$, or equivalently, of codimension 1, only possibly shifted from the origin by a vector. That is, for any $\boldsymbol{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$, the following set is a hyperplane in $\mathbb{R}^d$:*

$$\mathcal{H} = \left\{ \boldsymbol{x} \in \mathbb{R}^d \colon \boldsymbol{w}^\top \boldsymbol{x} + b = 0 \right\} \tag{2.1}$$

Definition 5 (Linearly Separable Dataset)**.** *Let $\mathcal{D} = (\boldsymbol{x}_i, y_i)$ be a dataset with $y_i \in \{+1, -1\}$. We say that $\mathcal{D}$ is linearly separable if there is a hyperplane $(\boldsymbol{w}, b)$ such that*

$$y_i(\boldsymbol{w}^\top \boldsymbol{x}_i + b) > 0, \quad \forall (\boldsymbol{x}_i, y_i) \in \mathcal{D}. \tag{2.2}$$

Remark 6. *In low dimensional spaces, the linearly separable condition may not hold. In high dimensional spaces, on the other hand, data points tend to be far away from each other so that it almost always hold.*

Given a linearly separable dataset, the perceptron algorithm aims to find a hyperplane that separates one class from the other. For convenience, we usually use a trick to absorb the term b. Instead of using $\boldsymbol{x}_i$ and $\boldsymbol{w}$, we use

$$\boldsymbol{x}_i \leftarrow \begin{pmatrix} \boldsymbol{x}_i \\ 1 \end{pmatrix} \quad \text{and} \quad \boldsymbol{w} \leftarrow \begin{pmatrix} \boldsymbol{w} \\ 1 \end{pmatrix}. \tag{2.3}$$

Then, the hyperplane becomes $\mathcal{H} = \left\{ \boldsymbol{x} \in \mathbb{R}^d \colon \boldsymbol{w}^\top \boldsymbol{x} = 0 \right\}$ and the linearly separable condition Eq. (2.2) becomes $y_i \left(\boldsymbol{w}^\top \boldsymbol{x}_i \right) > 0$.

2.2 Algorithm

The perceptron algorithm is an iterative algorithm that finds a hyperplane that separates one class from the other. It is described in Algorithm 2.2.

Algorithm 2.2.1 Perceptron

Initialize: Learnable parameter $\boldsymbol{w} = \mathbf{0}$

Output: Updated parameter $\boldsymbol{w}$

while True do
  $m = 0$ ▷ # of misclassified points
  for $(\boldsymbol{x}, y) \in \mathcal{D}$ **do**
    if $y(\boldsymbol{w}^\top \boldsymbol{x}) \leq 0$ **then** ▷ when $(\boldsymbol{x}, y)$ lies in the wrong side
      $\boldsymbol{w} \leftarrow \boldsymbol{w} + y\boldsymbol{x}$
      $m \leftarrow m + 1$
    end if
  end for
  if $m = 0$ **then**
    break ▷ If there are no misclassified points, training is done.
  end if
end while

In the algorithm, if a data point $(\boldsymbol{x}, y)$ is not classified well, the weight vector $\boldsymbol{w}$ is updated as follows: if $y = +1$ and $\boldsymbol{w}^\top \boldsymbol{x} \leq 0$, then $\boldsymbol{w}^\top \boldsymbol{x}$ increases because

$$(\boldsymbol{w} + \boldsymbol{x})^\top \boldsymbol{x} = \boldsymbol{w}^\top \boldsymbol{x} + \boldsymbol{x}^\top \boldsymbol{x} > \boldsymbol{w}^\top \boldsymbol{x}, \tag{2.4}$$

and if $y = -1$ and $\boldsymbol{w}^\top \boldsymbol{x} \geq 0$, then $\boldsymbol{w}^\top \boldsymbol{x}$ decreases because

$$(\boldsymbol{w} - \boldsymbol{x})^\top \boldsymbol{x} = \boldsymbol{w}^\top \boldsymbol{x} - \boldsymbol{x}^\top \boldsymbol{x} < \boldsymbol{w}^\top \boldsymbol{x}. \tag{2.5}$$

Note that $\boldsymbol{x} \neq 0$ because there is a nonzero entry in the last coordinate. Therefore, we can see that the update equation $\boldsymbol{w} \leftarrow \boldsymbol{w} + y\boldsymbol{x}$ is a way to make all data pairs $(\boldsymbol{x}, y)$ satisfy $y\,(\boldsymbol{w}^\top \boldsymbol{x}) > 0$.

2.3 Convergence of the algorithm

In this section, we prove the convergence of the perceptron algorithm. Without loss of generality, we assume that $\|\boldsymbol{w}^*\| = 1$. Since the size of the dataset is finite, we may also assume that $\|\boldsymbol{x}_i\| \leq 1$ for all $(\boldsymbol{x}_i, y_i) \in \mathcal{D}$ by rescaling. Let γ be the distance between the hyperplane and the closest point, that is,

$$\gamma = \min_{(\boldsymbol{x}_i, y_i) \in \mathcal{D}} |\, \boldsymbol{x}_i^\top \boldsymbol{w}^*| > 0. \tag{2.6}$$

This number γ is also known as the margin. This assumptions can be illustrated as in Fig. 2.3.

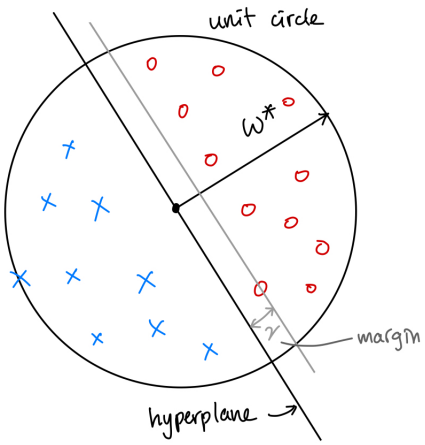


Figure 2.1: Assumptions of Perceptron

Theorem 7. *The perceptron algorithm makes at most* $1/\gamma^2$ *mistakes.*

Recall that we make an update $\boldsymbol{w} \leftarrow \boldsymbol{w} + y\boldsymbol{x}$ when we misclassified, i.e., $y\,(\boldsymbol{w}^\top \boldsymbol{x}) \leq 0$. Let $\boldsymbol{w}$ be a vector during optimization while $\boldsymbol{w}^*$ is the optimal hyperplane. Assume that a pair $(\boldsymbol{x}, y) \in \mathcal{D}$ is misclassified. We want to take a look at the change of two values: $\boldsymbol{w}^\top \boldsymbol{w}^*$ and $\boldsymbol{w}^\top \boldsymbol{w}$.

- The value $\boldsymbol{w}^\top \boldsymbol{w}^*$:

$$(\boldsymbol{w} + y\boldsymbol{x})^\top \boldsymbol{w}^* = \boldsymbol{w}^\top \boldsymbol{w}^* + y(\boldsymbol{w}^*)^\top \boldsymbol{x} \geq \boldsymbol{w}^\top \boldsymbol{w}^* + \gamma \tag{2.7}$$

  We can see the value $\boldsymbol{w}^\top \boldsymbol{w}^*$ grows by at least γ per each update.

- The value $\boldsymbol{w}^\top \boldsymbol{w}$:

$$(\boldsymbol{w} + y\boldsymbol{x})^\top (\boldsymbol{w} + y\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{w} + 2y\boldsymbol{w}^\top \boldsymbol{x} + y^2 \boldsymbol{x}^\top \boldsymbol{x} \leq \boldsymbol{w}^\top \boldsymbol{w} + 1 \tag{2.8}$$

  We can see the value $\boldsymbol{w}^\top \boldsymbol{w}$ grows by at most 1 per each update.

After M updates, since $\boldsymbol{w}$ was initialized to the zero, we obtain that

$$M\gamma \leq \boldsymbol{w}^\top \boldsymbol{w}^* = |\boldsymbol{w}^\top \boldsymbol{w}^*| \leq \|\boldsymbol{w}\| \cdot \|\boldsymbol{w}^*\| = \|\boldsymbol{w}\| = \sqrt{\boldsymbol{w}^\top \boldsymbol{w}} \leq \sqrt{M}. \tag{2.9}$$

Therefore, we have $M \leq 1/\gamma^2$, implying that the iteration terminates in finite number of steps. Notice that the number of steps only depends on the margin.

2.4 XOR problem

XOR (exclusive OR) problem is a classic problem in AI. In 1969 a famous book entitled *Perceptrons* by Minksy and Papert showed that it was impossible for perceptrons to learn an XOR function. Single layer perceptrons are capable of learning *linearly* separable patterns, that is, it is capable of separating data points with a single line. However, many data, including XOR inputs, are not linearly separable.

3 Multi-layer Perceptron

3.1 Gradient Descent Method

3.2 Algorithm

3.3 Vanishing Gradient Problem

4 More about Deep Learning

4.1 Batch Normalization (BN)

Batch Normalization [5] is introduced accelerate deep learning training by reducing internal covariate shift.

4.1.1 Transformation and Train

The transformation is described in Algorithm 4.2.1.

4.1.2 Inference Phase

Fix an activation x. Let $\{\mathcal{B}_i\}_{i=1}^N$ be the collection of mini-batches corresponding to x and $\mu_{\mathcal{B}_i}$ be the sample mean of $\mathcal{B}_i$. If training is done with the mini-batches, then

$$\mathbb{E}[x]_{(N)} = \frac{1}{N}\sum_{i=1}^{N}\mu_{\mathcal{B}_i} \tag{4.1}$$

Algorithm 4.1.1 Batch Normalizing Transform, applied to activation x over a mini-batch.

Input: Values of x over a minibatch: $\mathcal{B} = \{x_i\}_{i=1}^m$; Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}_{i=1}^m$

1: $\mu_{\mathcal{B}} \leftarrow \dfrac{1}{m}\sum_{i=1}^{m} x_i$ ▷ mini-batch mean

2: $\sigma_{\mathcal{B}}^2 \leftarrow \dfrac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$ ▷ mini-batch variance

3: $\hat{x}_i \leftarrow \dfrac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$ ▷ normalize

4: $y_i \leftarrow \gamma\hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$ ▷ scale and shift

will be used during inference since there is no mini-batch concept. If we train once more with mini-batch $\mathcal{B}_{N+1}$, then

$$E[x]_{(N+1)} = \frac{1}{N+1}\sum_{i=1}^{N+1}\mu_{\mathcal{B}_i} \tag{4.2}$$

will be used during inference. We compute $\mathbb{E}[x]_{(N+1)}$ using moving average technique:

$$\begin{aligned} E[x]_{(N+1)} &= \frac{1}{N+1}\sum_{i=1}^{N+1}\mu_{\mathcal{B}_i} \\ &= \frac{1}{N+1}\left(N\cdot E[x]_{(N)} + \mu_{\mathcal{B}_{N+1}}\right) \\ &= \frac{N}{N+1}E[x]_{(N)} + \frac{1}{N+1}\mu_{\mathcal{B}_{N+1}}. \end{aligned} \tag{4.3}$$

Likewise, the variance $Var[x]$ is computed using moving average technique, that is,

$$\begin{aligned} Var[x]_{(N+1)} &= \frac{m}{m-1}\left(\frac{1}{N+1}\sum_{i=1}^{N+1}\sigma_{\mathcal{B}_i}^2\right) \\ &= \frac{m}{m-1}\left[\frac{1}{N+1}\left(N\cdot\frac{m-1}{m}\cdot Var[x]_{(N)} + \sigma_{\mathcal{B}_{N+1}}^2\right)\right] \\ &= \frac{N}{N+1}Var[x]_{(N)} + \frac{m}{m-1}\cdot\frac{1}{N+1}\sigma_{\mathcal{B}_{N+1}}^2 \end{aligned} \tag{4.4}$$

where m is the batch size. The term $\frac{m}{m-1}$ is included to make the estimator unbiased. For each node, there are four parameters which are γ, β, moving mean, and moving variance. Scale parameter γ and shift parameter β are learnable/trainable while the others are not.

4.1.3 Practical Batch Normalization

In Tensorflow, there is another argument, which is called a 'momentum'. In practice, instead of computing Eq. 4.3 and Eq. 4.4, we set another hyperparameter and compute them as

Algorithm 4.1.2 Batch Normalizing Transform, applied to activation x over a mini-batch.

Input: Network $\boldsymbol{N}$ with trainable parameters Θ; subset of activations $\{x^{(k)}\}_{k=1}^{K}$

Output: Batch-normalized network for inference, $\boldsymbol{N}_{\text{BN}}^{\text{inf}}$

1: $\boldsymbol{N}_{\text{BN}}^{tr} \leftarrow \boldsymbol{N}$ ▷ Training BN network

2: **for** $k = 1 \dots K$ **do**

3: Add transformation $y^{(k)} = \text{BN}_{\gamma^{(k)},\beta^{(k)}}(x^{(k)})$ to $\boldsymbol{N}_{\text{BN}}^{tr}$ (Alg. 4.2.1

4: Modify each layer in $\boldsymbol{N}_{\text{BN}}^{tr}$ with input $x^{(k)}$ to take $y^{(k)}$ instead

5: **end for**

6: Train $\boldsymbol{N}_{\text{BN}}^{tr}$ to optimize the parameters $\Theta \cup \{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^{K}$

7: $\boldsymbol{N}_{\text{BN}}^{\text{inf}} \leftarrow \boldsymbol{N}_{\text{BN}}^{tr}$ ▷ Inference BN network with frozen parameters

8: **for** $k = 1, \dots, K$ **do** ▷ For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc.

9: Process multiple training mini-batches $\mathcal{B}$, each of size m, and average over them:

$$\mathbb{E}[x] \leftarrow \mathbb{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$$
$$Var[x] \leftarrow \frac{m}{m-1}\mathbb{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

10: In $\boldsymbol{N}_{\text{BN}}^{\text{inf}}$, replace the transform $y = \text{BN}_{\gamma,\beta}(x)$ with

$$y = \frac{\gamma}{\sqrt{Var[x] + \epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\mathbb{E}[x]}{\sqrt{Var[x] + \epsilon}}\right)$$

11: **end for**

follows:

$$E[x]_{(N+1)} = E[x]_{(N)} \cdot \eta + \mu_{\mathcal{B}_{N+1}} \cdot (1 - \eta) \tag{4.5}$$
$$Var[x]_{(N+1)} = Var[x]_{(N)} \cdot \eta + \sigma_{\mathcal{B}_{N+1}}^2 \cdot (1 - \eta) \tag{4.6}$$

where η is a momentum, it is set to be 0.99 (default).

4.1.4 Batch Normalization and Activation Function

Here, we consider an appropriate location of a batch normalization layer. In neural networks, we typically focus on transforms that consist of an affine transformation followed by an element-wise nonlinearity:

$$\boldsymbol{z} = g(W\boldsymbol{u} + \boldsymbol{b}) \tag{4.7}$$

where $\boldsymbol{W}$ and $\boldsymbol{b}$ are learnable parameters of the model and $g(\,\cdot\,)$ is an activation function. It is appropriate to add the BN transform immediately before the nonlinearity, by normalizing

$x = Wu + b$. Then $\boldsymbol{z} = g(W\boldsymbol{u} + \boldsymbol{b})$ is replaced with

$$\boldsymbol{z} = g(\mathbf{BN}(W\boldsymbol{u})) \tag{4.8}$$

where **BN** is applied independently to each dimension of $\boldsymbol{x} = W\boldsymbol{u}$. Note that the bias b can be ignored since its effect will be canceled by the subsequent mean subtraction, and since it does not affect the variance.

4.1.5 Batch-Normalized Convolutional Networks

For convolutional layers, we additionally want the normalization to obey the convolutional property. Let's consider a feature map of size $H \times W \times C$. Choose the ith channel of the feature map, so that its size will be $H \times W$, where $(i = 1, 2, \ldots, C)$. $H \times W$ different elements are normalized in the same way. That is, we learn a pair of parameters $\gamma^{(i)}$ and $\beta^{(i)}$ per channel, rather than per activation.Of course, there are two more non-trainable parameters, moving mean and variance, per channel.

4.2 Dropout

4.3 Weight Initialization

References

[1] Bernhard Schölkopf and Alexander J. Smola, *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*, The MIT Press (2001).

[2] V. Vapnik, *Principles of Risk Minimization for Learning Theory*, NIPS 1991.

[3] Olivier Chapelle, Jason Weston, Léon Bottou, and Vladimir Vapnik, *Vicinal Risk Minimization*, NIPS 2000.

[4] Hongyi Zhang et al., *mixup: Beyond Empirical Risk Minimization*, ICLR 2018.

[5] Sergey Ioffe and Christian Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, ICML 2015.