Chapter 7

Bias-Variance Trade-off, Dropout, Batch-Normalization

Reading

1. Bishop 1.3, 3.2, 14.2-14.3
2. Goodfellow 5.1-5.4, 7.1-7.3
3. Dropout Srivastava et al. (2014)
4. Batch-Normalization Ioffe and Szegedy (2015)

In this chapter, we will take our first look at how machine learning classifiers generalize to new data. We will first discuss the so-called Bias-Variance Tradeoff which indicates that the variance of the predictions of a model can be reduced by increasing its bias. Regularization is a technique to give us control over this tradeoff. We will then see a few popular regularization techniques, in particular two that are important in deep learning called Dropout and Batch-Normalization.

7.1 Bias-Variance Decomposition

Ideally, we want a classifier that accurately captures the regularity in the data but also works well for unseen data. Turns out this is typically impossible to both simultaneously. We will introduce this using regression.

Given our dataset $D = \left\{(x^i, y^i)\right\}_{i=1,\dots,n}$ we fit a model $f(x; w) \in \mathcal{F}$ where $\mathcal{F}$ is some class of models, say all neural networks with a given architecture; we will keep the dependence of f on w implicit in this section because we don't need it. We use a loss $\ell(f(x), y) = |f(x) - y|^2$ to fit this model by minimizing

$$\hat{R}(f) = \frac{1}{n} \sum_{i=1}^{n} |f(x^i) - y^i|^2 \tag{7.1}$$

This is of course the training loss, also called the empirical risk. A classifier that minimizes $\hat{R}(f)$ works well on the training data. If we want to measure how well a model works on new data from the distribution P we are interested in the the *population risk*

$$\begin{aligned} R(f) &= \int |f(x) - y|^2 \; P(x, y) \; \mathrm{d}x \; \mathrm{d}y \\ &= \mathop{\mathbb{E}}_{x} \left[\int |f(x) - y|^2 \; P(y|x) \; \mathrm{d}y \right]. \end{aligned} \tag{7.2}$$

It turns out that because the loss is quadratic, we can write down the minimizer of the population risk, formally, as

$$f^*(x) = \mathop{\mathbb{E}}_{y} [y|x]. \tag{7.3}$$

In other words, the optimal regressor is the conditional expectation of the targets y given a datum x. Since we do not know the data distribution P, we cannot compute the model f^*. We now compare some regression f that we may have obtained by minimizing (7.1) with the optimal f^*.

Observe that

$$\begin{aligned} (f(x) - y)^2 &= (f(x) - f^*(x) + f^*(x) - y)^2 \\ &= (f(x) - f^*(x))^2 + 2\,(f(x) - f^*(x))\,(f^*(x) - y) + (f^*(x) - y)^2. \end{aligned}$$

Substitute this expression in (7.2) to get

$$R(f) = \mathop{\mathbb{E}}_{x} \left[\int (f(x) - f^*(x))^2 \right] + \mathop{\mathbb{E}}_{(x,y) \sim P} \left[(f^*(x) - y)^2 \right] \tag{7.4}$$

Observe that the cross-term

$$\mathop{\mathbb{E}}_{x} \left[\int 2(f - f^*)(f^* - y) P(y|x) \; \mathrm{d}y \right] = 0$$

vanishes because $f^*(x) = \mathbb{E}[y|x] = \int y P(y|x) \mathrm{d}y$. In the first term, there is no y because the distribution $P(y|x)$ when integrated with respect to y is 1. The decomposition in (7.4) is insightful. The first term tells us how far our model $f(x)$ is from the optimal $f^*(x)$. The second term tells us how much the optimal model itself is from the data (x, y). The second term is not under our control because it does not depend on $f(x)$ at all. This term

$$\text{Bayes error} = \mathop{\mathbb{E}}_{(x,y) \sim P} \left[(f^*(x) - y)^2 \right]. \tag{7.5}$$

is irreducible error of any classifier f. It is only zero if the data (x, y) is coming from a deterministic source, i.e., there is no noise in the true targets y created by Nature and Nature's model (it is important to realize that this model is *not* f^*) is deterministic.

⚠ You can think of the Bayes error as being non-zero if the sensor used to measure y is noisy, there is no way we can get deterministic data in that case. If on the other hand the sensor is perfect, e.g., a large number of humans are annotating data very carefully like we often do in modern machine learning, the Bayes error is essentially zero.

We will now investigate the first term better. Notice that the model f is created using a finite dataset. Let us emphasize it as

$$f(x; D)$$

and rewrite the first term in (7.4) as

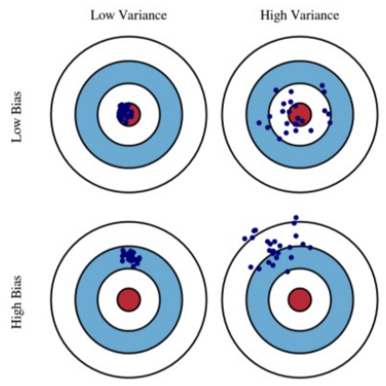
$$\begin{aligned}(f(x;D) - f^*(x))^2 &= \Big(f(x;D) - \mathop{\mathbb{E}}_D[f(x;D)] + \mathop{\mathbb{E}}_D[f(x;D)] - f^*(x)\Big)^2\\ &= \Big(f(x;D) - \mathop{\mathbb{E}}_D[f(x;D)]\Big)^2\\ &+ \Big(\mathop{\mathbb{E}}_D[f(x;D)] - f^*(x)\Big)^2\\ &+ 2\Big(f(x;D) - \mathop{\mathbb{E}}_D[f(x;D)]\Big)\ \Big(\mathop{\mathbb{E}}_D[f(x;D)] - f^*(x)\Big).\end{aligned}$$

Recall that the dataset is a random variable as well, it is a bunch of draws from the joint distribution P. Effectively, $f(x;D)$ which is our fitted model is a random variable that depends on the randomness of D. We now take the expectation over the *dataset* D on both sides of this equation.

$$\mathop{\mathbb{E}}_D\left[(f(x;D) - f^*(x))^2\right] = \underbrace{\mathop{\mathbb{E}}_D\left[\Big(\mathop{\mathbb{E}}_D[f(x;D)] - f^*(x)\Big)^2\right]}_{(\text{bias})^2} + \underbrace{\mathop{\mathbb{E}}_D\left[\Big(f(x;D) - \mathop{\mathbb{E}}_D[f(x;D)]\Big)^2\right]}_{\text{variance}}. \tag{7.6}$$

The cross-term again vanishes when we take the expection over the dataset. The first term is called the squared bias: it is the gap between the predictions of our model compared to the optimal model f^* created across many experiments each with a different dataset D. The second term is the variance and it measures how sensitive the model $f(x;D)$ to getting a particular dataset D to train on, if it is very sensitive a model fitted on D does not work well on most others datasets and consequently the variance is large. We will parse these quantities further soon.

⚠ Here is a good mnemonic to remember. Imagine the center of the bull's eye as the optimal classifier f^* and our darts as the model $f(x;D)$.



We have therefore shown that

$$R(f) = (\text{bias})^2 + \text{variance} + \text{Bayes error} \tag{7.7}$$

Recall that we want to minimize the population risk $R(f)$. We cannot do much about the Bayes error. If the model $f(x;D)$ is large and is fitted very well

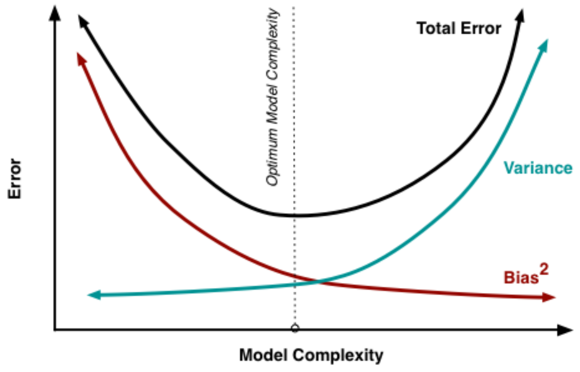


Figure 7.1: Population risk as a function of model capacity

on the dataset D, i.e., if its predictions match true y (notice that the optimal models predictions f^* are also close to y), the gap between the predictions of the fitted model and the optimal model is small on the dataset D. In other words, if our model is large we will have a small bias. The bias of a model decreases as we consider larger models $f(x; D)$. If our dataset is small, the model $f(x; D)$ is likely to have a large variance because it has not seen a large amount of data. The effect increases for larger models because they may use a larger number of nuisances i.e., features that are not relevant to prediction of targets. We call this over-fitting.

If we plot a picture of how the bias and variance change as model capacity (you can think of capacity simply as the number of parameters in a model for now) increases, we see a famous U-shaped curve for the sum of squared bias and variance shown in Figure 7.1. Given a dataset D we should pick a model that lies at the bottom of this curve to get a good population risk; this model makes a good tradeoff between bias and variance.

The caveat is that we do not have access to a lot of different datasets to measure the bias or the variance. This is why the bias-variance trade-off, although fundamental in machine learning/statistics and a great thinking tool, is of limited direct practical value.

7.1.0.1 Bias-variance tradeoff for classification

We have only talked about the bias-variance trade-off for regression. The development for classification is not very different and same principles hold. We first define an optimal classifier

$$f^*(x) = \underset{f \in \mathcal{F}}{\text{argmin}} \underset{(x,y)\sim P}{\mathbb{E}} [\ell(y, f(x))]$$

for a loss function ℓ. The bias, variance of a given classifier $f(x; D)$ relative to this optimal classifier and the Bayes error are given by

$$\begin{aligned} \text{bias} &= \underset{x}{\mathbb{E}} \, [\ell(f^*(x), f(x; D))] \\ \text{variance} &= \underset{D}{\mathbb{E}} [\ell(f(x; D), f^{\text{avg}}(x))] \\ \text{Bayes error} &= \underset{(x,y)\sim P}{\mathbb{E}} [\ell(y, f^*(x))] \,. \end{aligned} \tag{7.8}$$

where $f^{\text{avg}}(x) = \text{argmin}_f \, \mathbb{E}_D [\ell(y, f(x))]$; under the MSE loss this is the average of predictions of regressors on different datasets, for the MAE loss this is the median of the predictions of models trained on different datasets, for the zero-one loss it is the most frequent prediction of models trained on different datasets. We again have a trade-off that is obtained by decomposing the population risk

$$\underset{(x,y)\sim P}{\mathbb{E}} \left[\underset{D}{\mathbb{E}} [\ell(y, f(x; D))] \right] = \text{bias } + c_2 \text{variance } + c_1 \text{Bayes error.}$$

where c_1, c_2 are constants. You can read more about this in Pedro (2000).

7.1.0.2 Double-descent

The surprising thing is that for deep networks, we do not see this classical bias-variance trade-off. The population risk looks like

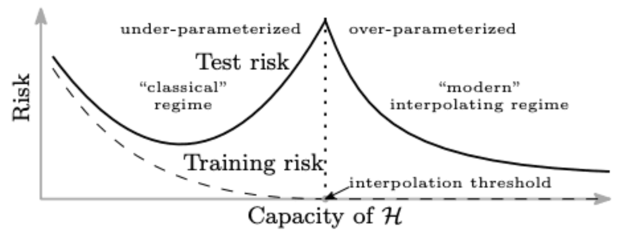


Figure 7.2: Double-descent curve: the validation error of deep networks decreases even if more and more complex models are fitted on the same data; there is no apparent over-fitting and growth in the variance of the classifier.

in what is now called the "double-descent" curve. The population risk of deep networks keeps decreasing even if we fit very large models on relatively small datasets, e.g., CIFAR-10 has 50,000 images, the model you will fit in HW 2 has about 1.6M weights and is considered a very small model by today's standards. We will see some heuristic derivation into why the population risk may look like this for deep networks but understanding this phenomenon which goes flat against established knowledge in machine learning is one of the big open problems in the study of deep networks today.

7.1.1 Cross-Validation

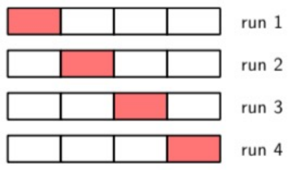
We have seen that the bias-variance trade-off requires us to consider multiple datasets. In practice, we only have *one* dataset that we collected by running an experiment. If this data is large, we can split it into two three parts

$$\text{data } = \text{training set } \cup \text{validation set } \cup \text{test set.}$$

The validation set is used to compare multiple models that we fit on the training set and pick the best performing one. This model is then run on the test set to demonstrate how well we have learned the data. The test set is necessary because across your design efforts to fit different models, you will evaluate on the validation set multiple times and this may lead to over-fitting on the validation set.

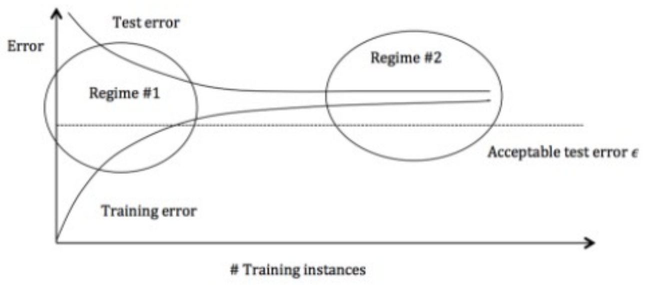
If the available data is not a lot, we want to use as much of the data as possible for training. If however only use a small fixed validation set for comparing models, we risk making mistakes in our choices. Cross-validation is a solution to this problem: it trains k different models, each time a fraction $(k-1)/k$ of the data is used as the training set and the remainder is used as the validation set. The validation performance of k models obtained by this process is averaged and used as a score to evaluate a particular model design (architecture, hyper-parameters etc).

⚠ 4-fold cross-validation.



7.1.1.1 Some practical tips

It is useful to think of the bias-variance trade-off when you fit deep networks in practice. If the training or test error is high, there are a number of ways to improve performance using the bias-variance tradeoff as a thinking tool.



In the first regime on the left, we have high validation error across cross-validation folds and low training error. This indicates that we have a high variance in the bias-variance trade-off. Typical techniques to counter this is to use a smaller model, get more data, or bagging a set of models together (will cover this in Section 7.3). In the second regime on the right, if the test error *and* the training error are close to each other but both are large, the model is likely to have high bias. In these cases, we should fit a more complex model (say increase the number of weights, or pick a different architecture), add more features to the training data (in the non-deep-learning setting) to give our model more discriminative features to use, or use boosting (we will cover this in Section 7.3).

7.1.1.2 Cautionary Tale

You will however notice that a lot of research papers in deep learning simply use validation data as test data. Their reasons for doing so are as follows. All researchers have the same large dataset from which they would create a potential test set, the researchers therefore also know the ground-truth labels of test images and it is difficult to trust them not to peek at the ground-truth labels to choose between models. If the test data is hidden from everyone, we need a centralized server for evaluating everyone's results. This is difficult because research is fundamentally about discovering new knowledge. Kaggle competitions or the ImageNet Challenge http://image-net.org/challenges/LSVRC are few instances where such a centralized server is available.

It is therefore debatable whether the current practice of using validation set as the test set should be considered valid. On the positive side, it makes results across different publications comparable to each other; if everyone reports the error of their model on the same validation set, it is easy to compare Algorithm A versus Algorithm B. On the negative side, this incentivizes extensive hyper-parameter tuning and risks results that are over-fitted on the validation data, e.g., new fields such as neural architecture search are particularly problematic in this context. This is also the main reason for the current "style of research" where folks judges the merit of machine learning research simply by checking whether Algorithm A gets better validation error than Algorithm B on standard datasets. This is not the correct way to do scientific research. The more appropriate way to instantiate the scientific method is to first formulate a hypothesis, e.g., is gene X correlated with cancer Y, then collect data that allows us to evaluate such an hypothesis and undertake appropriate statistical precautions report whether the hypothesis stands/does not stand.

That said, there are researchers who have evaluated others' claims (obtained using validation data, namely A better than B) on independent test data and reached similar conclusions, see for example https://arxiv.org/abs/1902.10811, so the evaluation methodology is broken but the progress is real.

7.2 Weight Decay

The set of models with smaller complexity are a subset of the set of models with larger complexity, e.g., if you are fitting a polynomial regression, you can consider the subset of models with coefficients of the higher-order terms equal to zero and have thus created the set linear regressors. Effectively, the space of *models* looks as follows.

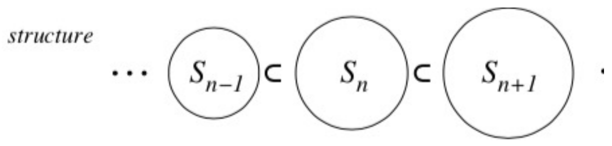


Figure 7.3: A cartoon of the space of models. The n in the picture refers to number of parameters in the model, not the number of data.

Let's say we are fitting a class of models with large complexity and are unsure whether the variance in the bias-variance trade-off will be large. We can either collect more data, or we can modify the loss function to encourage the training process to pick models of lower complexity.

Restricting the space of models that the training process searchers over to fit the data is called *regularization*. We will denote regularizers by

$$\text{regularizer} = \Omega(w)$$

and modify our loss function for fitting data to be

$$\ell'(w; x, y) := \ell(w; x, y) + \Omega(w).$$

Weight decay is one of the simplest regularization techniques and uses

$$\Omega(w) = \frac{\alpha}{2} \|w\|_2^2. \tag{7.9}$$

This is more widely known as ℓ_2 regularization because we use the ℓ_2 norm of the weights as the regularizer. It is also called Tikonov regularization, a name that comes from the literature on partial differential equations. The name weight decay comes from the neural networks literature of the 1980s. The gradient of the modified loss is

$$\nabla \ell'(w; x, y) = \nabla \ell(w; x, y) + \alpha\, w,$$

which gives

$$w^{t+1} = (1 - \eta\, \alpha) w^t - \eta \nabla \ell(w^t; x, y);$$

where η is the learning rate. in other words the weights w are encouraged to become smaller in magnitude when SGD takes a step using the negative gradient.

If we have a linear regression problem with $f(x; w) = w^\top x$ and X, Y are the matrices for the data and targets respectively, the regularized objective is

$$\frac{1}{2}\|Y - Xw\|_2^2 + \frac{\alpha}{2}\|w\|^2$$

and you can compute the minimizer by taking derivatives and setting them to zero to be

$$w^* = \left(X^\top X + \alpha I\right)^{-1} X^\top Y.$$

In other words, weight decay for linear regression adds elements to the diagonal of the data covariance matrix $X^\top X$. This results in a smaller inverse and thereby a smaller magnitude of w^*. Notice that if the covariance matrix is rank deficient, the regularized matrix is no longer rank deficient. If the covariance matrix has a large condition number (ratio of the largest and smaller eigenvalue), which makes taking the inverse numerically difficult, the regularized matrix has a better condition number.

7.2.1 Do not do weight decay on biases

If the input data and targets in linear regression are centered we do not need a bias parameter in our model. Notice however that if the dataset is not centered, the bias parameter is essential. Should we perform weight decay on the bias parameter in this case? The weight decay penalty prevents the bias parameter to adapt to the non-zero mean of the data. This is also important to keep in mind while training neural networks. We should not impose weight decay regularization on the bias parameters of the convolutional and fully-connected layers.

⚠ Weight decay is closely related to other norm-based penalties, e.g., ℓ_1 regularization sets

$$\Omega_{\ell_1}(w) = \alpha\|w\|_1.$$

As we discussed briefly in Chapter 6, such a regularizer encourages the weights to become sparse. Sparsity penalties are very common in the signal processing literature (e.g., compressed sensing, phase retrieval problems) but they are less common in the deep learning literature.

7.2.2 Maximum a posteriori (MAP) Estimation

MAP estimation gives a Bayesian perspective to regularization in machine learning. In maximum likelihood (ML) estimation, we were interested in solving for weights that maximize the likelihood of the observed data:

$$w^*_{\text{MLE}} = \underset{w}{\text{argmin}} -\frac{1}{n}\sum_{i=1}^{n} \log p_w(y^i | x^i; w).$$

MAP estimation enforces some prior knowledge we may have about the weights w. In Bayesian statistics, such prior knowledge is represented as a probability distribution, known as the *prior*, on the parameters w *before we see any data in the training process*, i.e., *a priori probability*

$$\text{prior} = p(w)$$

MAP estimation is regularized ML estimation. Given a prior distribution, we can use Bayes law to find the *posterior distribution* on the weights after observing the data

$$p(w|D) = \frac{p(D|w)\, p(w)}{p(D)} \tag{7.10}$$

Remember that the left hand side is a legitimate probability distribution with the denominator given by

$$Z := p(D) = \int p(D|w)\, p(w)\, \mathrm{d}w.$$

The denominator Z called the "evidence" or the partition function lies at the heart of all statistics, we will see why in Module 4.

MAP estimation finds the weights that maximize this *a posteriori* probability

$$\begin{aligned} w^*_{\text{MAP}} &= \operatorname*{argmax}_w \{\log p(D; w) + \log p(w)\} \\ &= -\frac{1}{n}\sum_{i=1}^n \log p_w(y^i|x^i; w) + \Omega(w) + \log Z(D) \\ &= -\frac{1}{n}\sum_{i=1}^n \log p_w(y^i|x^i; w) + \Omega(w). \end{aligned} \tag{7.11}$$

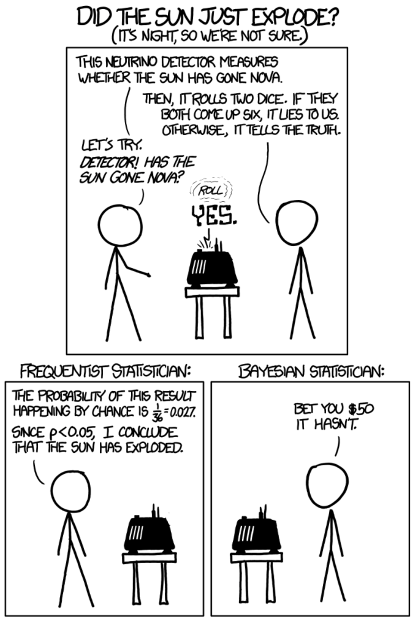
In the second step, we have denoted the log-prior by Ω

$$\log \text{prior}(w) := \Omega(w).$$

The final step follows because $Z(D)$ is not a function of the weights w and we can therefore can be ignored in the optimization.

7.2.2.1 Frequentist vs. Bayesian point of view

This section was our first view into Bayesian probabilities, as opposed to frequentist methods where we estimate probabilities by counting how many times a certain event occurs across our experiments. Frequentist probabilities are not designed to handle all situations. For instance we may be interested in estimating the probability of a very unlikely event, say that of the sun going supernova. This event has of course not happened yet and a frequentist notion of probability where we repeat the experiment many times and estimate the probability as the fraction of times the event occurs is not appropriate. The Bayesian point of view provides a natural way to answer these questions and the key idea is to encode our belief that the sun cannot go supernova as a prior probability.


DID THE SUN JUST EXPLODE?
(IT'S NIGHT, SO WE'RE NOT SURE.)
THIS NEUTRINO DETECTOR MEASURES WHETHER THE SUN HAS GONE NOVA.
THEN, IT ROLLS TWO DICE. IF THEY BOTH COME UP SIX, IT LIES TO US. OTHERWISE, IT TELLS THE TRUTH.
LET'S TRY. DETECTOR! HAS THE SUN GONE NOVA?
ROLL
YES.
FREQUENTIST STATISTICIAN:
THE PROBABILITY OF THIS RESULT HAPPENING BY CHANCE IS 1/36 = 0.027. SINCE p<0.05, I CONCLUDE THAT THE SUN HAS EXPLODED.
BAYESIAN STATISTICIAN:
BET YOU $50 IT HASN'T.

An alternate way to think about this is that the weights w of a model are considered a fixed quantity that we are supposed to estimate in a frequentist setting. The likelihood $p(D; w)$ is used to compare different models w and if one wanted an estimate of how much error we are making in our estimate, we would compute the variance in the Bias-variance tradeoff namely, the variance of our estimate across different draws of the dataset D. In the Bayesian point of view, there is a single dataset D and the uncertainty of our estimate of w^* would be expressed as the variance of the posterior distribution $p(w|D)$ in Bayes law.

7.2.2.2 Weight decay regularization is MAP estimation with Gaussian prior

Weight decay can be seen as using a Gaussian prior

$$p_{\text{weight-decay}}(w) \propto e^{-\frac{\|w\|_2^2}{(2\alpha^{-1})}}.$$

This is a multi-variate Gaussian distribution with mean zero and a diagonal covariance matrix with α^{-1} on the diagonal. The denominator is a function of α^{-1} and we do not need to worry about it while performing MAP estimation because it does not depend on w.

In other words, we have seen that weight decay in the training objective can be thought of as a MAP estimation using a Gaussian prior instead of ML estimation.

The Gaussian prior captures our a priori estimate of the true weights: the probability of the weights w being large is low (it is distributed as a Gaussian/Normal distribution). The likelihood term fits the weights to the data but instead of relying completely on the data which may result in a large variance (in cases when data is few), we also rely on the prior while fitting the model. This reasoning is captured in Bayes law.

Similarly, a sparsity penalty is MAP estimation with a Laplace prior For scalar random variables, the Laplace distribution is given by

$$p(w) = \frac{1}{2b} e^{-\frac{|x-\mu|}{b}}.$$

If we have

$$\Omega(w) = \|w\|_1$$

we can see that regularized ML, i.e., MAP estimation corresponds to using a Laplace prior on the weights w.

7.3 Dropout

We will next look at a very peculiar regularization technique that is unique to deep networks. Consider a two-layer network given by

$$\hat{y} = v^\top \text{dropout}\left(\sigma\left(S^\top x\right)\right).$$

Dropout is an operation that is defined as

$$\text{dropout}_{1-p}(h) = h \odot r \tag{7.12}$$

⚠ It is important to remember that a new dropout mask r is chosen for every input in the mini-batch.

where $r \in \{0,1\}^p$ is a binary mask and the notation $\odot$ denotes element multiplication. Each element of this mask r_k is a Bernoulli random variable with probability $1-p$

$$r_k = \begin{cases} 0 & \text{with probability } p \\ 1 & \text{with probability } 1-p. \end{cases}$$

In simple words, dropout takes the input activations h and zeros out a random subset of these; on an average p fraction of the activations are set to zero and the rest are kept to their original values. In pictures, it looks as follows.

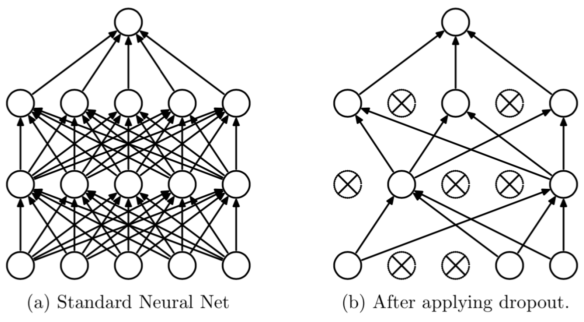


Figure 7.4: Dropout picks a random sparse subnetwork of a large deep network using the mask.

❓ The dropout mask is chosen at random for each image. Let us imagine that we have one dropout layer after every fully-connected layer. For the network shown in the figure with two hidden layers and 5 neurons at each layer, how many distinct sparse networks could be chosen for each input if $p = 0.5$?

The default Dropout probability is $p = 0.5$ in PyTorch, i.e., about half of the activations are set to zero for each input. Although you will see a lot of online code and architectures with this default value, you should experiment with the value of p, different values often given drastically different training and validation errors.

7.3.1 Bagging classifiers

Bagging, which is short for *bootstrap aggregation*, can be explained using a simple experiment. Suppose we wanted to estimate the average height μ of people in the world. We can measure the height of N individuals and obtain *one* estimate of the mean μ. This is of course unsatisfying because we know that our answer is unlikely to be the mean of the entire population. Bootstrapping computes multiple estimates of the mean μ_k over many *subsets* of the data N and reports the answer as

$$\mu := \text{mean}(\mu_k) + \text{stddev}(\mu_k).$$

Each subset of the data is created by sampling the original data with N samples *with replacement*. This is among the most influential ideas in statistics (Efron, 1992) because it is a very simple and general procedure to obtain the uncertainty of the estimate.

Effectively, the standard deviation of our new bootstrapped estimate of the mean is simply the standard deviation in the Bias-Variance trade-off with the big difference that we created multiple datasets D by sub-sampling with replacement of the original dataset.

Bagging is a classical technique in machine learning (Breiman, 1996) that trains multiple predictive models $f(x; w^k)$ for $k \in \{1, \dots, M\}$, one each for bootstrapped versions of the training dataset $\left\{D^1, \dots, D^M\right\}$. We aggregate the outputs of all these models together to form a *committee*

$$f(x; w^1, \dots, w^M) = \frac{1}{M} \sum_{k=1}^{M} f(x; w^k).$$

You can see that this procedure reduces the sum of the squared-bias and variance of the model (the first term in (7.4)) in the bias-variance trade-off by a factor of M if the errors with respect to the optimal classifier f^* of all the models $\{w^k\}$ are zero-mean and uncorrelated. In other words, the average error of a model can be reduced by a factor of M by simply averaging M versions of the model.

Bagging is always a good idea to keep in your mind. The winners of most high-profile machine learning competitions, e.g., the Netflix Prize (https://en.wikipedia.org/wiki/Netflix_Prize) or the ImageNet challenge, have been bagged classifiers created by fitting multiple architectures on the same dataset. Even today, random forests are among the most popular algorithms in the industry; these are ensembles of hundreds of models called decision trees on bootstrapped versions of data. A lot of times times, if we are combining diverse architectures in to the committee, we do not even need to bootstrap the data. Bagging does not work when the errors of the different models are not uncorrelated; this is however easy to fix by censoring out features in addition to boostrapping like it is done while training a random forests.

7.3.2 Some insight into how dropout works

Consider the following, very heuristic but nevertheless beautiful, argument in the original paper on dropout (Srivastava et al., 2014).

We will remove the nonlinearities and consider only a single layer linear model with dropout directly applied to the input layer $f(x; v) = v^\top \text{dropout}(x)$. Linear regression minimize the objective $\|y - Xw\|_2^2$ and similarly the dropout version of linear regression for our model would minimize

$$\min_w \mathop{\mathbb{E}}_R \left[\|y - (R \odot X)w\|_2^2 \right] \tag{7.13}$$

where each row of the matrix R consist of the dropout mask for the i^{th} row x^i of the data matrix X. Think carefully about the expectation over R on the outside, since we choose a random dropout mask each time an input is presented to SGD, the correct way to write dropout is using this expectation over the masks. Each entry of R is a Bernoulli random variable with probability $1 - p$ of being 1. Note that

$$\mathop{\mathbb{E}}_R [R \odot X] = (1-p)X$$

and the $(ij)^{\text{th}}$ element is

$$\left(\mathop{\mathbb{E}}_R \left[(R \odot X)^\top (R \odot X) \right] \right)_{ij} = \begin{cases} (1-p)^2 \left(X^\top X \right)_{ij} & \text{if } i \neq j \\ (1-p) \left(X^\top X \right)_{ii} & \text{else.} \end{cases}$$

We can use these two expressions to compute the objective in (7.13) to be

$$\mathop{\mathbb{E}}_R [\|y - (R \odot X)w\|_2] = \|y - (1-p)Xw\|^2 + \underbrace{p(1-p)w^\top \text{diag}(X^\top X)w}_{\Omega(w)}.$$

In other words, for linear regression, dropout is equivalent to weight-decay where the coefficient α in (7.9) depends on the diagonal of the data covariance and is different for different weights. If a particular data dimension varies a lot, i.e., $(X^\top X)_{ii}$ is large, dropout tries to squeeze its weight to zero. We can also absorb the factor of $1 - p$ into the weights w to get

$$\mathop{\mathbb{E}}_R [\|y - (R \odot X)w\|_2] = \|y - X\tilde{w}\|^2 + \underbrace{\left(\frac{p}{1-p} \right) \tilde{w}^\top \text{diag}(X^\top X)\tilde{w}}_{\Omega(w)} \tag{7.14}$$

where $\tilde{w} = (1-p)w$. This makes the regularization more explicit, if $p \approx 0$, most activations are retained by the mask and regularization is small.

Next, bagging provides a very intuitive understanding of how dropout works in a deep network at test time. We now write out the classifier explicitly as

$$f(x; w, r^k) = \sum_{i=1}^{d} w_k \left(x_k \odot r_i^k \right);$$

note that the mask r^k is not a parameter of the model, we have simply chosen to make it more explicit for the sequel. We now imagine each mask as creating a *bootstrapped* version of the model; different masks r^k give different classifiers even if the weights v and the input x is the same for all.

⚠ Training with dropout is equivalent to introducing weight decay on the weights. Remember however that this argument is only rigorous for linear regression models (the derivation essentially remains the same for matrix factorization). This connection of dropout with weight decay will also be apparent in Module 4 when we look at how to train a Bayesian deep network.

It is important to realize that there is no subsampling of training dataset happening here like classical boosting; we are instead forming multiple models by adding randomness to how the input is propagating through the deep network. For a linear classifier this is equivalent because

$$\sum_{i=1}^{d} w_k \left(x_k \odot r_i^k \right) = \sum_{i=1}^{d} \left(w_k \odot r_i^k \right) x_k =: f(x; w^k);$$

we can either mask out the input or mask the weights and think of the masked weights w^k as a new model.

Remark 1. You will often see folks in the literature say that dropout is regularizes by preventing co-adaptation of the neurons at each hidden layer. The motivation for this statement is that the weights of the succeeding layer cannot fixate too much upon a particular feature at the input because the feature can be zeroed out by the dropout mask. This prevents too much specialization of neurons in the hidden layer and ensures that the prediction is made using a large number of diverse features, not just a few specific ones. This is not a rigorous argument but it is a reasonable argument in view of the experiments of Hubel and Wiesel (see http://centennial.rucares.org/index.php?page=Neural_Basis_Visual_Perception). The human brain is robust to large parts of it going missing/inhibited.

Bagging is expensive at test time, it involves having to compute the predictions of all the models in the committee. In the case of dropout, in this linear heuristic argument, we can compute the committee prediction to be

$$\begin{aligned} f(x; w) &= \frac{1}{M} \sum_{k=1}^{M} \sum_{i=1}^{d} \left(w_k \odot r_i^k \right) x_k \\ &= \sum_{i=1}^{d} \left(w_k \odot \frac{1}{M} \sum_{k=1}^{M} r_i^k \right) x_k \\ &\approx \sum_{i=1}^{d} \left(w_k \odot (1-p) \right) x_k. \end{aligned} \tag{7.15}$$

This is very fortunate, it indicates that given weights w of a model trained using dropout, we can compute the *committee average* over models created using dropout masks simply by scaling the weights by a factor $1-p$. This should not be surprising, after all the equivalent training objective in (7.14) has $\tilde{w} = (1-p)w$ as the effective weights of the weights. Another important point to note is that there is no masking of activations at test time.

Although the argument in this section works only for linear models, we will bravely extend the intuition to deep networks.

7.3.3 Implementation details of dropout

The recipe for using dropout is simple: (i) the activations at the input of each dropout layer are zeroed out using a Bernoulli random variable of probability $1-p$ (the PyTorch layer takes the probability of zeroing out activations as argument which is p in our derivations; (ii) at test time, the weights of layers immediately preceding dropout are scaled by a factor of $1-p$ to compute the predictions of the "committee".

Inverted Dropout. It is cumbersome to remember the parameter p that was used for training at test time. Deep learning libraries use a clever trick: they simply scale the output activations of dropout layer by $1/(1-p)$ during training. Training or testing the modified model using dropout gives an extra factor of $(1-p)$ like (7.14) and (7.15) respectively and therefore the final model can be used as is without any further scaling of the weights or activations.

The operation `model.train()` in PyTorch sets the model in the training mode. This is a null-operation and does not do anything for fully-connected, convolutional, softmax etc. layers. For the dropout later, it sets a boolean variable in the layer that samples the Bernoulli mask for all the input activations and scales the output activations by $1/(1-p)$. The complementary operation is `model.eval()` in PyTorch which you should use to set the model in evaluation mode. This is again a null-operation for other layers but for the dropout layer, it resets this boolean variable to indicate that no Bernoulli masks should be sampled and no masking should be performed.

7.3.4 Using dropout as a heuristic estimate of uncertainty

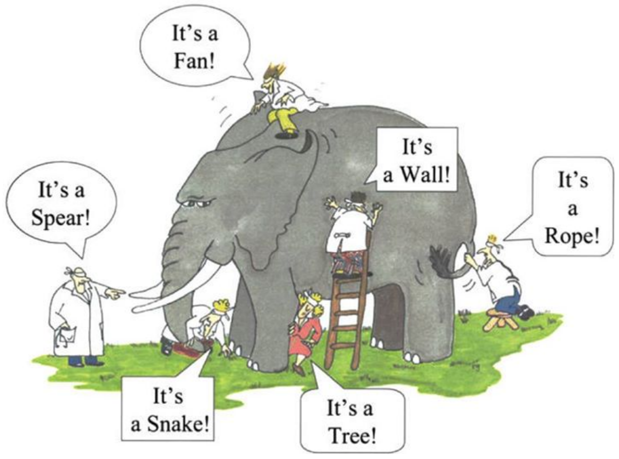
We can extend the motivation from bagging to use dropout as a cheap heuristic to get an estimate of the uncertainty of the prediction at test time. Suppose we use dropout at test time just like we do it at training time, i.e., each time one test input is presented to the deep network, we sample multiple Bernoulli masks $r^1, \ldots, r^M$ and compute multiple predictions for the same test input

$$\left\{ f(x; w, r^1), \ldots, f(x; w, r^M) \right\}.$$

The variance of these predictions can be used as heuristic of the uncertainty of the deep network while making predictions on the test input x. This is an estimate of the so-called *aleatoric or statistical uncertainty*. It captures our understanding that the weights w of a trained deep network are inherently uncertain and different training experiments, in particular, different masks r^k will give rise to different weights. The variance across a few sampled masks thus indicates how uncertain the model is about its predictions. Dropout is a neat and cheap trick for this purpose; it is quite commonly used in this fashion in medical applications where it is important to not only predict the outcome but also characterize the uncertainty of this prediction. We will see more powerful ways to compute aleatoric uncertainty in Module 4.

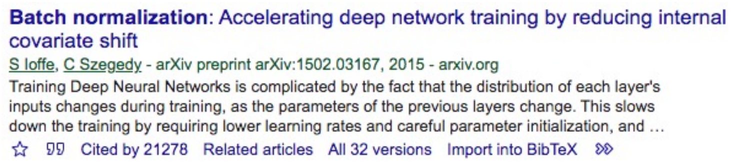
Remark 2. Broadly speaking, the connection of dropout with weight decay is contentious. If it were rigorous, we should be able to get the same performance as dropout by using appropriate weight decay (this is a good idea for the course project!). In practice, the validation error using dropout is very good and cannot be achieved by tweaking weight decay. Another aspect is that since we would like to average over lots of dropout masks in the training process, networks with dropout should be trained for many more iterations of SGD than networks without dropout to get the same training error. The benefit is that the test error is much better for dropout. What exactly dropout does is a subject of some mystery and there are other alternative explanations (e.g., Bayesian dropout in Module 4).

Our understanding of dropout is no different than that of these blind scientists trying to identify an elephant.



7.4 Batch-Normalization

Batch-Normalization (BN) is another layer that is very commonly used in deep learning. BN is very popular with more than 20,000 citations in about 5 years.



Batch normalization: Accelerating deep network training by reducing internal covariate shift
S Ioffe, C Szegedy - arXiv preprint arXiv:1502.03167, 2015 - arxiv.org
Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and …
☆ 99 Cited by 21278 Related articles All 32 versions Import into BibTeX »

7.4.1 Covariate shift

Covariate shift is a common problem with real data. The experimental conditions under which training data was gathered are subtly different from the situation in which the final model is deployed. For instance, in cancer diagnosis the training set may have an over-abundance of diseased patients, often of a specific subtype endemic in the location where the data was gathered. The model may be deployed in another part of the world where this subtype of cancer is not that common.

The mis-match between training and test *data* distribution is called covariate shift. Even if the labels depend on some known way $y|x$ on the covariates, i.e., given the genetic features of a person x their likelihood of a cancer y is the same regardless of which part of the world the person is from, the fact that we do not have training data from the entire population of the world forces the classifier to be tested on a data distribution that is different from what it was trained for.

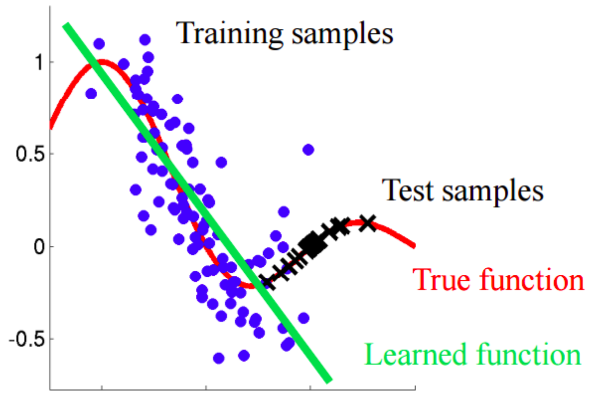


Figure 7.5: Covariate shift correction for a regression problem

Covariate shift is outside our fundamental assumption in Chapter 1 that training and test data come from the same distribution. It is however a problem that is often seen in practice and typical ways to counter it basically look as follows.

1. Train a classifier $\hat{w}$ on the available training data D.
2. Update the trained classifier using data from the test distribution $D' = \{(x^i, y^i)\}_{i=n+1,\ldots,n+m}$ in addition to the original training dataset

$$w^* = \underset{w}{\text{argmin}} \; \frac{1}{n+m} \sum_{i=1}^{n+m} p^i \, \ell^i(w) + \Omega(w - \hat{w}) \tag{7.16}$$

   where p^i is some weighing factor that indicates how similar the datum (x^i, y^i) is to the *test data distribution*. The regularization $\Omega(w - w^*)$ forces the new weights w^* to remain close to the old weights $\hat{w}$.

The above methods go under the umbrella of *doubly robust estimation*. We will not study it in this course. The results look similar to the ones shown in Figure 7.5.

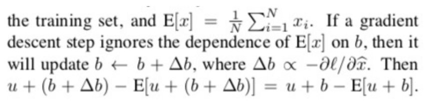
7.4.2 Internal covariate shift

If we are working under the standard machine learning assumption of test data drawn from the same distribution as the training data, then there is no covariate shift.

Recall that we whiten the inputs, say using Principal Component Analysis (PCA), for linear regression in order to decorrelate the input features; you can using a simple argument of how this changes the condition number of the data covariance matrix $X^\top X$ and accelerates the convergence of gradient descent using a calculation similar to the final problem in HW 2.

Deep networks are like any other model in this aspect and whitening of the inputs is also beneficial; the ZCA transform (or Mahalanobis whitening) is a close cousin of PCA and usually works better for image-based data. It is natural to expect that since each layer of a deep network takes the activations of the preceding layer as input, we should whiten the activations before the computation in the layer. The authors of the BN paper came upon an interesting through what is clearly a mistake. Their reasoning was based on a simple example: if we have a mini-batch of inputs $\left\{x^1, \ldots, x^{\mathscr{b}}\right\}$ and our layer simply adds a learnable bias b to this

⚠ This is the mistake in the original BN paper.


the training set, and $\mathrm{E}[x] = \frac{1}{N}\sum_{i=1}^{N} x_i$. If a gradient descent step ignores the dependence of $\mathrm{E}[x]$ on b, then it will update $b \leftarrow b + \Delta b$, where $\Delta b \propto -\partial \ell / \partial \widehat{x}$. Then $u + (b + \Delta b) - \mathrm{E}[u + (b + \Delta b)] = u + b - \mathrm{E}[u + b]$.

$$h = x + b.$$

If this layer whitens its output before passing it on to the next layer, we will have

$$\hat{h} := h - \frac{1}{\mathscr{b}} \sum_{i=1}^{\mathscr{b}} x^i.$$

The output $\hat{h}$ does not depend on the bias b. They argued, incorrectly, that the back-propagation update of the bias $\overline{b}$ is equal to $\overline{\hat{h}}$. This is not true of course because

$$\overline{b} = \overline{\hat{h}}\, \frac{\mathrm{d}\hat{h}}{\mathrm{d}b} = 0$$

in our notation where $\overline{h} = \mathrm{d}\ell / \mathrm{d}h$. Nevertheless the motivation of the batch-normalization operation is sound, we would like to whiten the input activations to each layer of a deep network.

Batch-Normalization is a technique for whitening the output activations of each layer in a deep network.

Naively, this would involve computing expressions of the form

$$\hat{h} = (\mathrm{Cov}(h))^{-1/2} \left(h - \frac{1}{\mathscr{b}} \sum_{i=1}^{\mathscr{b}} h^i \right).$$

This is not easy to do because the features are high-dimensional vectors, the covariance matrix $\mathrm{Cov}(h)$ is a very large matrix. This makes computing $\hat{h}$ difficult for every mini-batch. Nevertheless, whitening helps and here is how it is done in the batch-normalization module:

$$\hat{h} = \frac{h - \mathbb{E}(\{h^1, \ldots, h^{\mathscr{b}}\})}{\sqrt{\mathrm{Var}(\{h^1, \ldots, h^{\mathscr{b}}\}) + \epsilon}}. \tag{7.17}$$

The constant ϵ in the denominator prevents $\hat{h}$ from becoming very large in magnitude if the variance is small for a particular mini-batch. It is important to note that both the expectation and the variance are computed for every feature. Let us make this clear: if $h \in \mathbb{R}^{\mathscr{b} \times p}$, i.e., p features for this layer, the $i^{\text{th}} \in \{1, \ldots, \mathscr{b}\}$ input of the mini-batch and the $j^{\text{th}} \in \{1, \ldots, p\}$ of the feature for $\hat{h}$ is given by

$$\hat{h}_{ij} = \frac{\hat{h}_{ij} - \frac{1}{\mathscr{b}} \sum_{i=1}^{\mathscr{b}} h_{ij}}{\sqrt{\mathrm{Var}(\{h_{1j}, h_{2j}, \ldots, h_{\mathscr{b}j})\}}}.$$

Let us give names to these parameters

$$\begin{aligned}\mathbb{R}^p \ni \mu &= \mathbb{E}(\{h^1, \ldots, h^{\not{b}}\}) \\ \mathbb{R}^p \ni \sigma^2 &= \mathrm{Var}(\{h^1, \ldots, h^{\not{b}}\}).\end{aligned} \tag{7.18}$$

The authors of the original BN paper decided that mere normalization may not be enough, e.g., if you normalize the activations *after a sigmoid activation*, the layer may essentially become linear because the activations are prevented from going too far to the right or too far too the left of the origin. This brings the second key idea in BN, that of affine scaling of the output $\hat{h}$. The BN layer really implements two

$$\hat{h} = a \left(\frac{h - \mathbb{E}(\{h^1, \ldots, h^{\not{b}}\})}{\sqrt{\mathrm{Var}(\{h^1, \ldots, h^{\not{b}}\}) + \epsilon}} \right) + b. \tag{7.19}$$

where $a, b \in \mathbb{R}^p$, i.e., each feature has its own multiplier a and bias b. The final BN operation in short is therefore

$$\hat{h} = a \left(\frac{h - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) + b.$$

The affine scaling parameters a, b are the only parameters in BN that are updated using backpropagation. The mean μ and variance σ^2 are unique to every mini-batch and therefore do not have any backpropagation gradient.

Execute the following code in your Jupyter notebook and check how the BN layer is implemented in PyTorch

```
import torch.nn as nn
m = nn.BatchNorm1d(15)
print(m.weight, m.bias)
print(m.running_mean, m.running_var)
```

The weight and bias here are the affine scaling parameters; and running_mean, running_var are μ, σ^2 respectively. You will see that requires_grad is True only for the former.

7.4.2.1 BN for convolutional layers

The activations of a convolutional layer are a 4-dimensional matrix (or a tensor)

$$h \in \mathbb{R}^{\not{b} \times c \times w \times h}.$$

The distinction between convolutional layers compared to fully-connected layers is that the convolutional filter weights are shared for the whole input channel $w \times h$. We can therefore think of each *channel as a feature* and compute the BN mean and standard deviation over the batch dimension, as well as the width and height. In pseudo-code, this looks as follows.

```
# t is still the incoming tensor of shape [bb, c, w, H]
# but mean and stddev are computed along (0, 2, 3) axes and
have just [c] shape
mean = mean(t, axis=(0, 2, 3))
stddev = stddev(t, axis=(0, 2, 3))
for i in 0..bb-1, x in 0..h-1, y in 0..w-1:
    out[i,:,x,y] = normalize(t[i,:,x,y], mean, stddev)
```

7.4.2.2 Running updates of the mean and variance in BN

BN computes the statistics over mini-batches. Even if we trained a model using mini-batch updates we would still like to be able to use this model at test time with a single input; it may not always be possible to wait for a few test images to make predictions. The weights of the network are trained to work with whitened features so we definitely need some way to whiten the features of a test input, ignoring the whitening at test time will result in wrong predictions.

The BN layer solves this issue by maintaining a running average of the mean and variance statistics of mini-batches during training. Effectively, the buffers running_mean, running_var (note that these are not parameters/weights, they are not updated using backprop) are updated after *each mini-batch* during training as

$$\begin{aligned}\text{running\_mean}^{t+1} &= \rho\,\text{running\_mean}^{t} + (1-\rho)\,\mu\\ \text{running\_var}^{t+1} &= \rho\,\text{running\_var}^{t} + (1-\rho)\,\sigma^2.\end{aligned}$$

The parameter ρ is called a momentum parameter for the BN layer and makes sure that updates to running_mean/var are slow and one mini-batch cannot affect the stored value too much. Note that whitening is still performed at training time using μ, σ^2; we simply record the running average in the buffers running_mean/var. If model.train() is called, then the mini-batch statistics are used to whiten the features. If model.eval() is called, then the stored buffers running_mean/var are used to whiten the outputs.

7.4.2.3 How is all this related to internal covariate shift?

You might be surprised that nothing in this section is related to covariate shift that we discussed at the beginning. Let us try to understand heuristically why BN is said to help with internal covariate shift.

Each layer of a deep network treats its input activations as the data and predicts the output activations. As the weights of different layers are updated using backprop during training, the *distribution* of input activations keeps shifting. Effectively, each layer is constant suffering a covariate shift because the layers below it are updated and the weights of the top layers have to adapt to this shifting distribution. This is what is known as *internal covariate shift*. BN normalizes the output activations to approximately have zero mean and unit variance and therefore reduces the internal covariate shift.

⚠ There are many caveats with this heuristic argument. The main one is to observe that the backpropagation gradient of all layers is coupled, so it is not as if the layers are updated independently of each other and cause interval covariate shifts to the other layers; the updates of all the weights in the network are coupled and it is unclear why (or even if) internal covariate shift occurs.

7.4.3 Problems with batch-normalization

There are two big problems with BN.

1. The affine parameters are updated using backpropagation and small changes mini-batch statistics which can result in large changes to the whitened output $(h-\mu)/\sqrt{\sigma^2+\epsilon}$ will result in very large updates to a, b. This makes the affine parameters problematic when you train networks. In general, it is a good idea to first fit a model without the affine BN parameters, you can do so by using affine=False in nn.BatchNorm1d.

2. The mean and variance buffers of the BN layer are updated using runnings statistics of the per-mini-batch statistics. This does not affect

training because the statistics of each mini-batch are computed independently, but it does affect evaluation because the buffers are used to whiten the features of the test input. If the test input has slightly different pixel intensity statistics than the training image, then the BN buffers are not ideal for whitening and such images are classified incorrectly.

BN before ReLU or ReLU before BN

Should we apply BN before or after the nonlinearity? The purpose of a BN layer is to keep the activations close to zero in their mean and a standard-deviation of one. Imagine if we are using a ReLU nonlinearity after BN, about half of our features h have negative values which the rectification will set to zero. In this case the distribution of features given to the next layer is not zero-mean, unit-variance so we are not achieving our goal of whitening correctly. Further, it is possible that the bias parameter b in BN is negative in which case the activations could mostly be negative and ReLU will set all of them to zero and result in a large loss in information. On the other hand, if we have BN after ReLU, the input to the BN layer has a lot of zeros and we are now computing mean/variance over a number of sparse features; the mini-batch mean/variance estimated here may not be accurate therefore BN may not perform its job of correctly whitening its outputs. You can read more about similar problems at http://torch.ch/blog/2016/02/04/resnets.html

As you can see, BN is an incredibly intricate operation without necessarily sound theoretical foundation for all the moving parts. But it works, training a deep fully-connected network is very difficult without BN, and even for convolutional layers it often makes training insensitive to the choice of learning rate. You should think about BN very carefully in your implementations; a lot of problems of the kind, "I trained my model, it gives a good training error but very poor validation error", or "I am fine-tuning from this task, but get very poor validation error on a new task", or other problems in reinforcement learning, meta-learning, transfer learning etc. can be boiled down to an incorrect/inaccurate understanding of batch-normalization. This is further complicated by the interaction with other operations such as Dropout, e.g., see https://arxiv.org/abs/1801.05134. Studying the effect of BN in meta-learning/transfer-learning is a good idea for a course project.

How does Dropout affect BN?

Since dropout is active during training, the buffered statistics are the running mean/variance of the dropped out activations. Dropout is not used at test time, so the test time statistics, even for the same image can be quite different. A simple to solve this problem is to run the model in training model once on the validation set (without making weight updates using backpropagation) for the BN buffers to settle to their non-droppped out values and then compute the validation error; this usually results in a marignal improvement in the validation error.

Variants of BN

There are variants of batch-normalization that have cropped out to alleviate some of its difficulties. For instance, layer normalization (https://arxiv.org/abs/1607.06450) normalizes across the features instead of the mini-batch which makes it work better for small mini-batches. Another variant known as group-normalization computes the mean/variance estimate in BN across multiple partitions of the mini-batch which makes the result of group-normalization independent of the batch-size. These variants work in some cases and do not work in some cases and often the specific normalization is largely dependent on the problem domain, e.g., group normalization works better for image segmentation but layer normalization and batch-normalization do not so well there.

Bibliography

Breiman, L. (1996). Bagging predictors. *Machine learning*, 24(2):123–140.

Efron, B. (1992). Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics*, pages 569–593. Springer.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Pedro, D. (2000). A unified bias-variance decomposition and its applications. In *17th International Conference on Machine Learning*, pages 231–238.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.