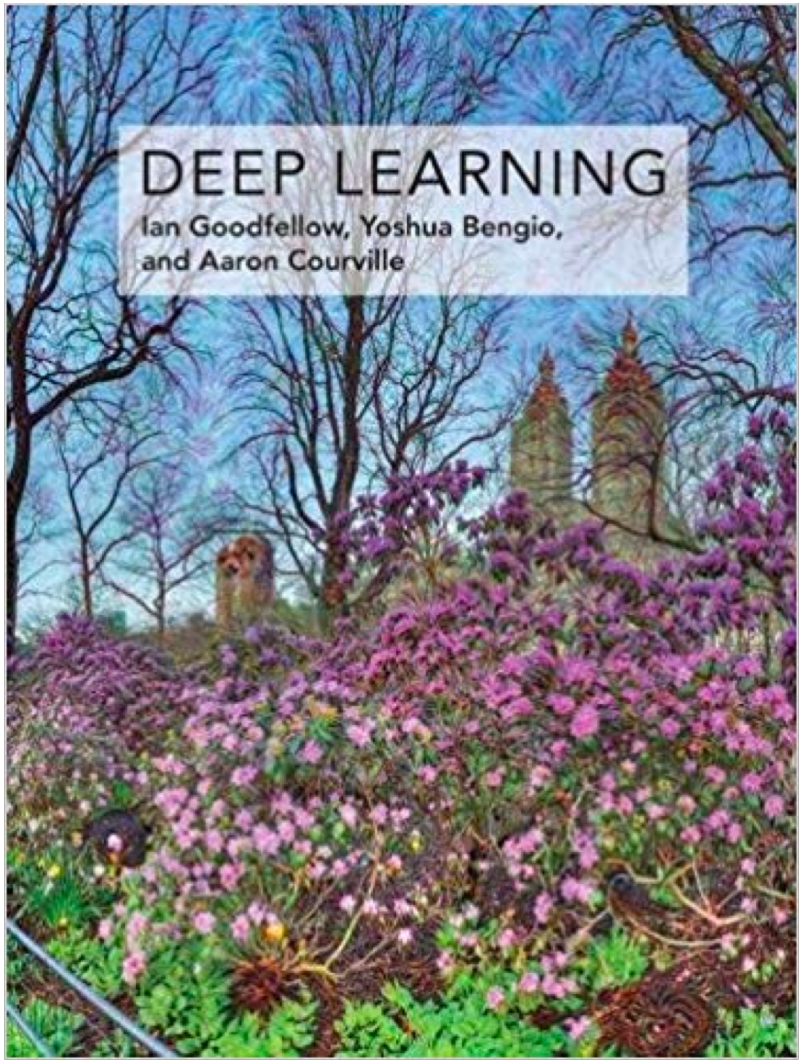
國立交通大學
National Chiao Tung University

Deep Learning
深度學習
Fall 2018

Regularization
(Chapter 7)



Prof. Chia-Han Lee
李佳翰 副教授

Regularization

- Many strategies used in machine learning are explicitly designed to reduce the test error, possibly at the expense of increased training error. These strategies are known collectively as regularization.

- In section 5.2.2, we defined regularization as “any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.”

- We almost always find that the best fitting model is a large model that has been regularized appropriately.

Parameter norm penalties

- Many regularization approaches are based on limiting the capacity of models by adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the objective function J.

- We denote the regularized objective function by $\tilde{J}$:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta}), \tag{7.1}$$

  where $\alpha \in [0, \infty)$ is a hyperparameter that weights the relative contribution of the norm penalty term, Ω, relative to the standard objective function J.

- Setting α to 0 results in no regularization. Larger values of α correspond to more regularization.

Parameter norm penalties

- We typically choose to use a parameter norm penalty Ω that penalizes only the weights of the affine transformation at each layer and leaves the biases unregularized.

- Each weight specifies how two variables interact. Fitting the weight well requires observing both variables in a variety of conditions.

- Each bias controls only a single variable. This means that we do not induce too much variance by leaving the biases unregularized. The biases typically require less data than the weights to fit accurately. Also, regularizing the bias parameters can introduce a significant amount of underfitting.

Parameter norm penalties

- In the context of neural networks, it is sometimes desirable to use a separate penalty with a different α coefficient for each layer of the network.

- Because it can be expensive to search for the correct value of multiple hyperparameters, it is still reasonable to use the same weight decay at all layers just to reduce the size of search space.

L^2 parameter regularization

- The L^2 parameter norm penalty is commonly known as weight decay. This regularization strategy drives the weights closer to the origin by adding a regularization term $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{\omega}\|_2^2$ to the objective function.

- In other academic communities, L^2 regularization is also known as ridge regression or Tikhonov regularization.

- We can gain some insight into the behavior of weight decay regularization by studying the gradient of the regularized objective function.

L^2 parameter regularization

- We assume no bias parameter, so $\boldsymbol{\theta}$ is just $\boldsymbol{\omega}$. Such a model has the following total objective function:

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2} \boldsymbol{w}^\top \boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}), \tag{7.2}$$

with the corresponding parameter gradient

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}). \tag{7.3}$$

- To take a single gradient step to update the weights, we perform this update:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon \left(\alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) \right). \tag{7.4}$$

Written another way, the update is

$$\boldsymbol{w} \leftarrow (1 - \epsilon\alpha)\boldsymbol{w} - \epsilon \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}). \tag{7.5}$$

- The addition of the weight decay term has modified the learning rule to multiplicatively shrink the weight vector by a constant factor on each step before gradient update.

L^2 parameter regularization

- This describes what happens in a single step. But what happens over the entire course of training?

- We further simplify the analysis by making a quadratic approximation to the objective function in the neighborhood of the value of the weights that obtains minimal unregularized training cost, $\boldsymbol{\omega}^* = \arg\min_{\boldsymbol{\omega}} J(\boldsymbol{\omega})$.

- If the objective function is truly quadratic, as in the case of fitting a linear regression model with mean squared error, then the approximation is perfect.

L^2 parameter regularization

- The approximation $\hat{J}$ is given by

$$\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^\top \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*), \tag{7.6}$$

  where $\boldsymbol{H}$ is the Hessian matrix of J with respect to $\boldsymbol{\omega}$ evaluated at $\boldsymbol{\omega}^*$.

- There is no first-order term in this quadratic approximation, because $\boldsymbol{\omega}^*$ is defined to be a minimum, where the gradient vanishes. Likewise, because $\boldsymbol{\omega}^*$ is the location of a minimum of J, we can conclude that $\boldsymbol{H}$ is positive semidefinite.

- The minimum of $\hat{J}$ occurs where its gradient

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*) \tag{7.7}$$

  is equal to 0.

L^2 parameter regularization

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*) \tag{7.7}$$

- To study the effect of weight decay, we modify equation 7.7 by adding the weight decay gradient. We can now solve for the minimum of the regularized version of $\hat{J}$. We use the variable $\tilde{\boldsymbol{\omega}}$ to represent the location of the minimum.

$$\alpha \tilde{\boldsymbol{w}} + \boldsymbol{H}(\tilde{\boldsymbol{w}} - \boldsymbol{w}^*) = 0 \tag{7.8}$$

$$(\boldsymbol{H} + \alpha \boldsymbol{I})\tilde{\boldsymbol{w}} = \boldsymbol{H}\boldsymbol{w}^* \tag{7.9}$$

$$\tilde{\boldsymbol{w}} = (\boldsymbol{H} + \alpha \boldsymbol{I})^{-1} \boldsymbol{H}\boldsymbol{w}^* \tag{7.10}$$

- As α approaches 0, the regularized solution $\tilde{\boldsymbol{\omega}}$ approaches $\boldsymbol{\omega}^*$. But what happens as α grows?

L^2 parameter regularization

- Because $\boldsymbol{H}$ is real and symmetric, we can decompose it into a diagonal matrix $\boldsymbol{\Lambda}$ and an orthonormal basis of eigenvectors $\boldsymbol{Q}$, such that $\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{\mathrm{T}}$.

- Applying the decomposition to equation 7.10, we obtain

$$\tilde{\boldsymbol{w}} = (\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{\top} + \alpha\boldsymbol{I})^{-1}\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{\top}\boldsymbol{w}^{*} \tag{7.11}$$

$$= \left[\boldsymbol{Q}(\boldsymbol{\Lambda} + \alpha\boldsymbol{I})\boldsymbol{Q}^{\top}\right]^{-1}\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{\top}\boldsymbol{w}^{*} \tag{7.12}$$

$$= \boldsymbol{Q}(\boldsymbol{\Lambda} + \alpha\boldsymbol{I})^{-1}\boldsymbol{\Lambda}\boldsymbol{Q}^{\top}\boldsymbol{w}^{*}. \tag{7.13}$$

- We see that the effect of weight decay is to rescale $\boldsymbol{\omega}^{*}$ along the axes defined by the eigenvectors of $\boldsymbol{H}$. Specifically, the component of $\boldsymbol{\omega}^{*}$ that is aligned with the i-th eigenvector of $\boldsymbol{H}$ is rescaled by a factor of $\frac{\lambda_i}{\lambda_i+\alpha}$.

L^2 parameter regularization

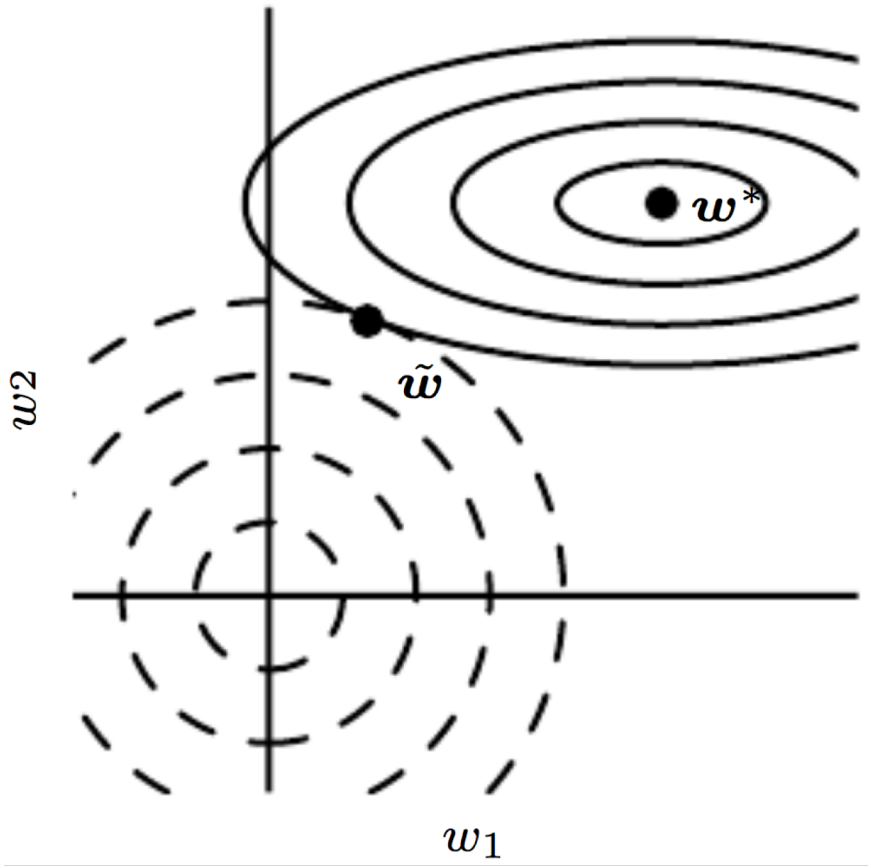
$$\boldsymbol{Q}(\boldsymbol{\Lambda} + \alpha \boldsymbol{I})^{-1}\boldsymbol{\Lambda}\boldsymbol{Q}^\top \boldsymbol{w}^*. \tag{7.13}$$

- Along the directions where the eigenvalues of H are relatively large, for example, where $\lambda_i \gg \alpha$, the effect of regularization is relatively small. Yet components with $\lambda_i \ll \alpha$ will be shrunk to have nearly zero magnitude.

- Only directions along which the parameters contribute significantly to reducing the objective function are preserved relatively intact. In directions that do not contribute to reducing the objective function, a small eigenvalue of the Hessian tells us that movement in this direction will not significantly increase the gradient.

- Components of the weight vector corresponding to such unimportant directions are decayed away through the use of the regularization throughout training.

L^2 parameter regularization



- The solid ellipses represent contours of equal value of the unregularized objective. The dotted circles represent contours of equal value of the L^2 regularizer.
- At the point $\tilde{\omega}$, these competing objectives reach an equilibrium. In the first dimension, the eigenvalue of the Hessian of J is small. The objective function does not increase much when moving horizontally away from ω^*. Because the objective function does not express a strong preference along this direction, the regularizer has a strong effect on this axis. The regularizer pulls ω_1 close to zero.
- In the second dimension, the objective function is very sensitive to movements away from ω^*. The corresponding eigenvalue is large, indicating high curvature. As a result, weight decay affects the position of ω_2 relatively little.

L^2 parameter regularization

- For linear regression, the cost function is the sum of squared errors:

$$(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\top (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}). \tag{7.14}$$

When we add L^2 regularization, the objective function changes to

$$(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^\top (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \frac{1}{2}\alpha \boldsymbol{w}^\top \boldsymbol{w}. \tag{7.15}$$

- This changes the normal equations for the solution from

$$\boldsymbol{w} = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y} \tag{7.16}$$

to

$$\boldsymbol{w} = (\boldsymbol{X}^\top \boldsymbol{X} + \alpha \boldsymbol{I})^{-1} \boldsymbol{X}^\top \boldsymbol{y}. \tag{7.17}$$

L^2 parameter regularization

- The matrix $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$ in equation 7.16 is proportional to the covariance matrix $\frac{1}{m}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$. Using L^2 regularization replaces this matrix with $\left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X} + \alpha\boldsymbol{I}\right)^{-1}$ in equation 7.17.

- The new matrix is the same as the original one, but with the addition of α to the diagonal. The diagonal entries of this matrix correspond to the variance of each input feature.

- L^2 regularization causes the learning algorithm to "perceive" the input $\boldsymbol{X}$ as having higher variance, which makes it shrink the weights on features whose covariance with the output target is low compared to this added variance.

L^1 regularization

- While L^2 weight decay is the most common form of weight decay, another option is to use L^1 regularization.

- Formally, L^1 regularization on the model parameter $\boldsymbol{\omega}$ is

$$\Omega(\boldsymbol{\theta}) = ||\boldsymbol{w}||_1 = \sum_i |w_i|, \tag{7.18}$$

  that is, as the sum of absolute values of the individual parameters.

- As with L^2 weight decay, L^1 weight decay controls the strength of the regularization by scaling the penalty Ω using a positive hyperparameter α.

L^1 regularization

- Thus, the regularized objective function $J(\boldsymbol{\omega}; \boldsymbol{X}, \boldsymbol{y})$ is

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha ||\boldsymbol{w}||_1 + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}), \quad (7.19)$$

with the corresponding gradient (actually, sub gradient)

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \text{sign}(\boldsymbol{w}) + \nabla_{\boldsymbol{w}} J(\boldsymbol{X}, \boldsymbol{y}; \boldsymbol{w}), \quad (7.20)$$

where $\text{sign}(\boldsymbol{\omega})$ is the sign of $\boldsymbol{\omega}$ applied element-wise.

- By inspecting equation 7.20, we can see that the regularization contribution to the gradient no longer scales linearly with each ω_i; instead it is a constant factor with a sign equal to $\text{sign}(\omega_i)$.

- One consequence of this form of the gradient is that we will not necessarily see clean algebraic solutions to quadratic approximations of $J(\boldsymbol{X}, \boldsymbol{y}; \boldsymbol{\omega})$ as we did for L^2 regularization.

L^1 regularization

- Our simple linear model has a quadratic cost function that we can represent via its Taylor series. Alternately, we could imagine that this is a truncated Taylor series approximating the cost function of a more sophisticated model.

- The gradient in this setting is given by

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*), \tag{7.21}$$

where, again, $\boldsymbol{H}$ is the Hessian matrix of J with respect to $\boldsymbol{\omega}$ evaluated at $\boldsymbol{\omega}^*$.

- Because the L^1 penalty does not admit clean algebraic expressions in the case of a fully general Hessian, we will further assume that the Hessian is diagonal, $\boldsymbol{H} = \text{diag}([H_{1,1}, \ldots, H_{n,n}])$, where each $H_{i,i} > 0$.

L^1 regularization

- This assumption holds if the data for the linear regression problem has been preprocessed to remove all correlation between the input features, which may be accomplished using PCA.

- Our quadratic approximation of the L^1 regularized objective function decomposes into a sum over the parameters:

$$\hat{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{w}^*; \boldsymbol{X}, \boldsymbol{y}) + \sum_i \left[\frac{1}{2} H_{i,i} (\boldsymbol{w}_i - \boldsymbol{w}_i^*)^2 + \alpha |w_i| \right]. \tag{7.22}$$

- The problem of minimizing this approximate cost function has an analytical solution (for each dimension i), with the following form:

$$w_i = \text{sign}(w_i^*) \max \left\{ |w_i^*| - \frac{\alpha}{H_{i,i}}, 0 \right\}. \tag{7.23}$$

L^1 regularization

- Consider the situation where $\omega_i^* > 0$ for all i. There are two possible outcomes:

1. The case where $\omega_i^* \leq \frac{\alpha}{H_{i,i}}$. Here the optimal value of ω_i under the regularized objective is simply $\omega_i = 0$. This occurs because the contribution of $J(\boldsymbol{\omega}; \boldsymbol{X}, \boldsymbol{y})$ to the regularized objective $\tilde{J}(\boldsymbol{\omega}; X, y)$ is overwhelmed—in direction i—by the L^1 regularization, which pushes the value of ω_i to zero.
2. The case where $\omega_i^* > \frac{\alpha}{H_{i,i}}$. In this case, the regularization does not move the optimal value of ω_i to zero but instead just shifts it in that direction by a distance equal to $\frac{\alpha}{H_{i,i}}$.

- A similar process happens when $\omega_i^* < 0$, but with the L^1 penalty making ω_i less negative by $\frac{\alpha}{H_{i,i}}$, or 0.

L^1 regularization

- In comparison to L^2 regularization, L^1 regularization results in a solution that is more sparse. Sparsity in this context refers to the fact that some parameters have an optimal value of zero.

- The sparsity property induced by L^1 regularization has been used extensively as a feature selection mechanism (choosing which subset of the available features should be used).

- The well known LASSO (least absolute shrinkage and selection operator) model integrates an L^1 penalty with a linear model and a least-squares cost function. The L^1 penalty causes a subset of the weights to become zero, suggesting that the corresponding features may safely be discarded.

Dataset augmentation

- The best way to make a machine learning model generalize better is to train it on more data. In practice, the amount of data we have is limited. One way to solve this problem is to create fake data and add it to the training set.

- This approach is easiest for classification. The main task facing a classifier is to be invariant to a wide variety of transformations. We can generate new $(\boldsymbol{x}, y)$ pairs easily just by transforming the $\boldsymbol{x}$ inputs in our training set.

- This approach is not as readily applicable to many other tasks. For example, it is difficult to generate new fake data for a density estimation task unless we have already solved the density estimation problem.

Dataset augmentation

- Dataset augmentation is particularly effective for a specific classification problem: object recognition.

- Operations like translating the training images a few pixels in each direction can often greatly improve generalization, even if the model has already been designed to be partially translation invariant. Many other operations, such as rotating the image or scaling the image, have also proved quite effective.

- One must be careful not to apply transformations that would change the correct class. For example, optical character recognition tasks require recognizing the difference between “b” and “d” and the difference between “6” and “9,” so horizontal flips and 180° rotations are not appropriate.

Dataset augmentation

- Injecting noise in the input to a neural network can also be seen as a form of data augmentation.
- For many classification and even some regression tasks, the task should still be possible to solve even if small random noise is added to the input. One way to improve the robustness of neural networks is simply to train them with random noise applied to their inputs.
- Noise injection also works when the noise is applied to the hidden units, which can be seen as doing dataset augmentation at multiple levels of abstraction.
- Dropout can be seen as a process of constructing new inputs by multiplying by noise.

Dataset augmentation

- To compare the performance of one machine learning algorithm to another, it is necessary to perform controlled experiments. When comparing machine learning algorithm A and machine learning algorithm B, make sure that both algorithms are evaluated using the same hand-designed dataset augmentation schemes.

Noise robustness

- For some models, the addition of noise with infinitesimal variance at the input of the model is equivalent to imposing a penalty on the norm of the weights. In the general case, noise injection can be much more powerful than simply shrinking the parameters, especially when the noise is added to the hidden units.

- Another way that noise has been used is by adding it to the weights. This technique has been used primarily in the context of RNN.

- Noise applied to the weights can also be interpreted as equivalent to a more traditional form of regularization, encouraging stability of the function to be learned.

Noise robustness

- Assume that with each input presentation we include a random perturbation $\epsilon_{\boldsymbol{W}} \sim N(\boldsymbol{\epsilon}; \mathbf{0}, \eta \boldsymbol{I})$ of the network weights. We use a standard l-layer MLP, and denote the perturbed model as $\hat{y}_{\epsilon_{\boldsymbol{W}}}(\boldsymbol{x})$. The objective function is

$$\tilde{J}_{\boldsymbol{W}} = \mathbb{E}_{p(\boldsymbol{x},y,\epsilon_{\boldsymbol{W}})}\left[(\hat{y}_{\epsilon_{\boldsymbol{W}}}(\boldsymbol{x}) - y)^2\right] \tag{7.31}$$

$$= \mathbb{E}_{p(\boldsymbol{x},y,\epsilon_{\boldsymbol{W}})}\left[\hat{y}^2_{\epsilon_{\boldsymbol{W}}}(\boldsymbol{x}) - 2y\hat{y}_{\epsilon_{\boldsymbol{W}}}(\boldsymbol{x}) + y^2\right]. \tag{7.32}$$

- For small η, the minimization of J with added weight noise (with covariance $\eta \boldsymbol{I}$) is equivalent to minimization of J with an additional regularization term: $\eta \mathbb{E}_{p(\boldsymbol{x},y)}[\|\nabla_{\boldsymbol{W}} \hat{y}(\boldsymbol{x})\|^2]$.

Noise robustness

- This form of regularization encourages the parameters to go to regions of parameter space where small perturbations of the weights have a relatively small influence on the output.

- In other words, it pushes the model into regions where the model is relatively insensitive to small variations in the weights, finding points that are not merely minima, but minima surrounded by flat regions.

- In the simplified case of linear regression (where, for instance, $\hat{y}(\boldsymbol{x}) = \boldsymbol{\omega}^{\mathrm{T}}\boldsymbol{x} + b$), this regularization term collapses into $\eta \mathbb{E}_{p(\boldsymbol{x})}[\|\boldsymbol{x}\|^2]$, which is not a function of parameters and therefore does not contribute to the gradient of $\tilde{J}_{\boldsymbol{W}}$ with respect to the model parameters.

Injecting noise at the output targets

- Most datasets have some number of mistakes in the y labels. It can be harmful to maximize $\log p(y|\boldsymbol{x})$. One way to prevent this is to explicitly model the noise on the labels. For example, we can assume that for some small constant ϵ, the training set label y is correct with probability $1 - \epsilon$, and otherwise any of the other possible labels might be correct.

- This assumption is easy to incorporate into the cost function analytically, rather than by explicitly drawing noise samples. For example, label smoothing regularizes a model based on a $\mathrm{softmax}$ with k output values by replacing the hard 0 and 1 classification targets with targets of $\frac{\epsilon}{k-1}$ and $1 - \epsilon$, respectively. The cross-entropy loss may then be used with these soft targets.

Injecting noise at the output targets

- Maximum likelihood learning with a softmax classifier and hard targets may actually never converge—the softmax can never predict a probability of exactly 0 or exactly 1, so it will continue to learn larger and larger weights, making more extreme predictions forever.

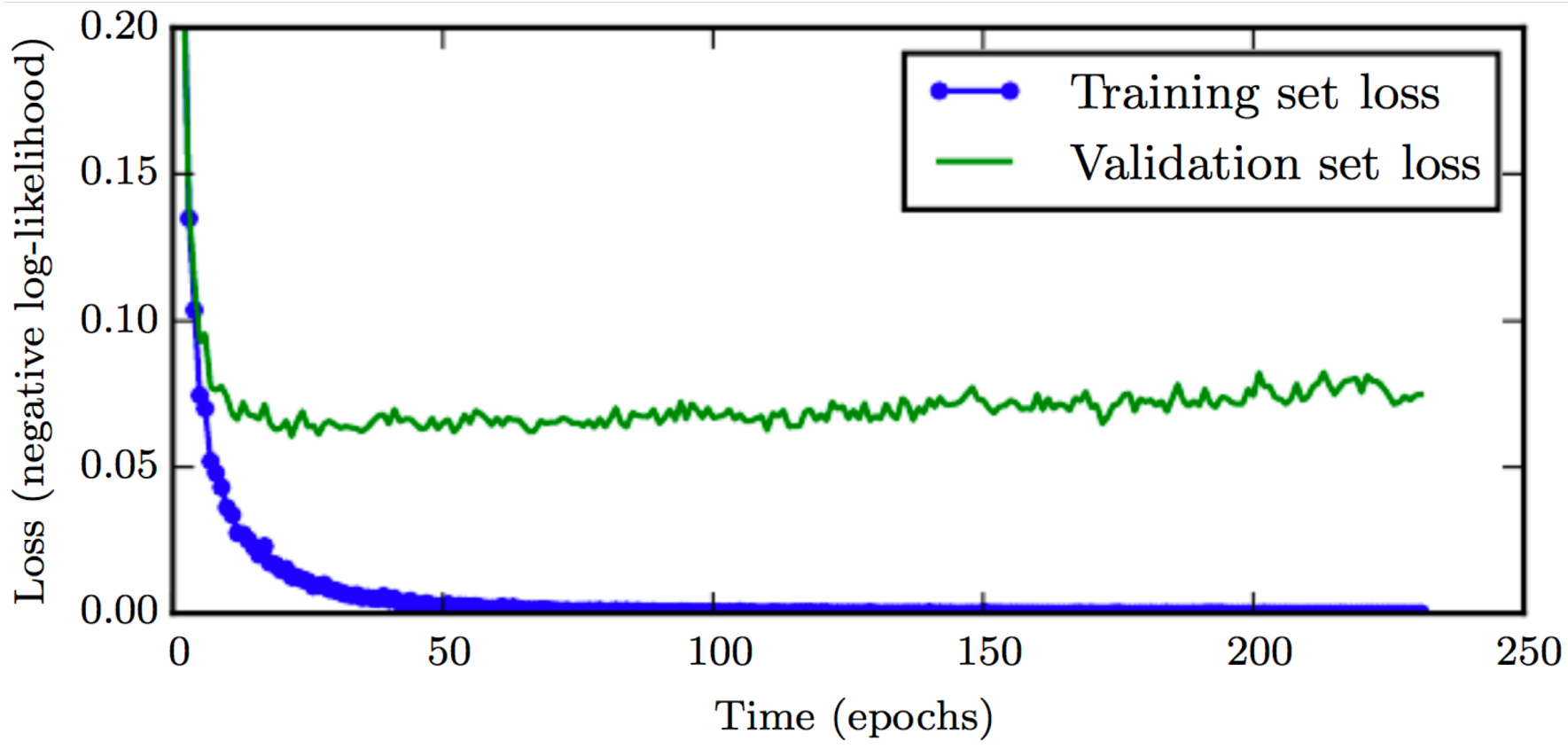
- It is possible to prevent this scenario using other regularization strategies like weight decay. Label smoothing has the advantage of preventing the pursuit of hard probabilities without discouraging correct classification.

Early stopping

- When training large models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, but validation set error begins to rise again. This means we can obtain a model with better validation set error (and thus, hopefully better test set error) by returning to the parameter setting at the point in time with the lowest validation set error.

- Every time the error on the validation set improves, we store a copy of the model parameters. When the training algorithm terminates, we return these parameters, rather than the latest parameters. The algorithm terminates when no parameters have improved over the best recorded validation error for pre-specified # of iterations.

Early stopping



Early stopping

- This strategy is known as early stopping. It is probably the most commonly used form of regularization in deep learning. Its popularity is due to both its effectiveness and its simplicity.

- One way to think of early stopping is as a very efficient hyperparameter selection algorithm. In this view, the number of training steps is just another hyperparameter. We can see in figure 7.3 that this hyperparameter has a U-shaped validation set performance curve.

- In early stopping, we are controlling the effective capacity of the model by determining how many steps it can take to fit the training set.

Early stopping

- The only significant cost to choosing the "training time" hyperparameter automatically via early stopping is running the validation set evaluation periodically during training. Ideally, this is done in parallel to the training process on a separate machine, separate CPU, or separate GPU from the main training process.

- An additional cost to early stopping is the need to maintain a copy of the best parameters. This cost is generally negligible, because it is acceptable to store these parameters in a slower and larger form of memory. Since the best parameters are written to infrequently and never read during training, these occasional slow writes have little effect on the total training time.

Early stopping

- Early stopping requires almost no change in the underlying training procedure, the objective function, or the set of allowable parameter values. This means that it is easy to use early stopping without damaging the learning dynamics. This is in contrast to weight decay, where one must be careful not to use too much weight decay and trap the network in a bad local minimum.

- Early stopping requires a validation set, which means some training data is not fed to the model. To best exploit this extra data, one can perform extra training after the initial training with early stopping has completed. In the second, extra training step, all the training data is included.

Early stopping

- There are two basic strategies one can use for this second training procedure.

- One strategy is to initialize the model again and retrain on all the data. In this second training pass, we train for the same number of steps as the early stopping procedure determined was optimal in the first pass.

- However, there is not a good way of knowing whether to retrain for the same number of parameter updates or the same number of passes through the dataset. On the second round of training, each pass through the dataset will require more parameter updates because the training set is bigger.

Early stopping

- Another strategy for using all the data is to keep the parameters obtained from the first round of training and then continue training, but now using all the data.

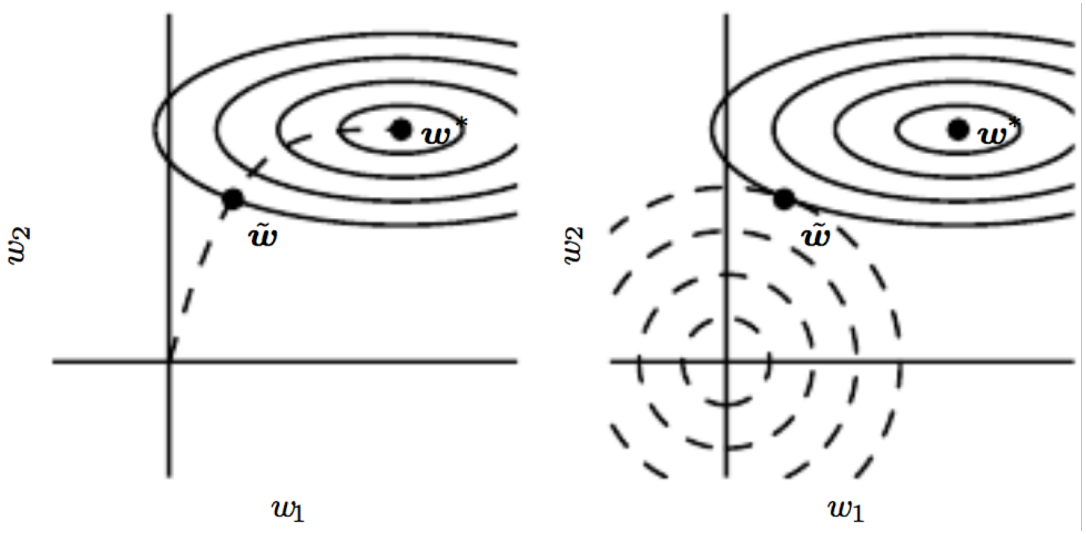
- At this stage, we now no longer have a guide for when to stop in terms of a number of steps. Instead, we can monitor the average loss function on the validation set and continue training until it falls below the value of the training set objective at which the early stopping procedure halted.

- This strategy avoids the high cost of retraining the model from scratch but is not as well behaved. For example, the objective on the validation set may not ever reach the target value, so this strategy is not even guaranteed to terminate.

Early stopping

- Early stopping also has the benefit of providing regularization without requiring the addition of penalty terms to the cost function or the computation of the gradients of such additional terms.

- Early stopping has the effect of restricting the optimization procedure to a relatively small volume of parameter space in the neighborhood of the initial parameter value $\boldsymbol{\theta}_o$.



Early stopping

- Imagine taking τ optimization steps (corresponding to τ training iterations) and with learning rate ϵ. We can view the product $\epsilon\tau$ as a measure of effective capacity. Assuming the gradient is bounded, restricting both the number of iterations and the learning rate limits the volume of parameter space reachable from $\boldsymbol{\theta}_o$. In this sense, $\epsilon\tau$ behaves as if it were the reciprocal of the coefficient used for weight decay.

- We can show how early stopping is equivalent to L^2 regularization (in the case of a simple linear model with a quadratic error function and simple gradient descent). We examine a simple setting where the only parameters are linear weights ($\boldsymbol{\theta} = \boldsymbol{\omega}$).

Early stopping

- We can model the cost function J with a quadratic approximation in the neighborhood of the empirically optimal value of the weights $\boldsymbol{\omega}^*$:

$$\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^\top \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*), \tag{7.33}$$

where $\boldsymbol{H}$ is the Hessian matrix of J with respect to $\boldsymbol{\omega}$ evaluated at $\boldsymbol{\omega}^*$. Given the assumption that $\boldsymbol{\omega}^*$ is a minimum of $J(\boldsymbol{\omega})$, we know that $\boldsymbol{H}$ is positive semidefinite.

- Under a local Taylor series approximation, the gradient is given by

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*). \tag{7.34}$$

Early stopping

- We are going to study the trajectory followed by the parameter vector during training. For simplicity, let us set the initial parameter vector to the origin, that is $\boldsymbol{\omega}^{(0)} = \mathbf{0}$. Let us study the approximate behavior of gradient descent on J by analyzing gradient descent on $\hat{J}$:

$$\boldsymbol{w}^{(\tau)} = \boldsymbol{w}^{(\tau-1)} - \epsilon \nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}^{(\tau-1)}) \tag{7.35}$$

$$= \boldsymbol{w}^{(\tau-1)} - \epsilon \boldsymbol{H}(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*), \tag{7.36}$$

$$\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^* = (\boldsymbol{I} - \epsilon \boldsymbol{H})(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*). \tag{7.37}$$

- Let us now rewrite this expression in the space of the eigenvectors of $\boldsymbol{H}$, exploiting the eigendecomposition of $\boldsymbol{H}$: $\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{\mathrm{T}}$, where $\boldsymbol{\Lambda}$ is a diagonal matrix and $\boldsymbol{Q}$ is an orthonormal basis of eigenvectors.

$$\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^* = (\boldsymbol{I} - \epsilon \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top)(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*) \tag{7.38}$$

$$\boldsymbol{Q}^\top(\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^*) = (\boldsymbol{I} - \epsilon \boldsymbol{\Lambda})\boldsymbol{Q}^\top (\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*) \tag{7.39}$$

Early stopping

- Assuming that ϵ is chosen to be small enough to guarantee $|1 - \epsilon\lambda_i| < 1$, the parameter trajectory during training after τ parameter updates is as follows:

$$\boldsymbol{Q}^\top \boldsymbol{w}^{(\tau)} = [\boldsymbol{I} - (\boldsymbol{I} - \epsilon\boldsymbol{\Lambda})^\tau]\boldsymbol{Q}^\top \boldsymbol{w}^*. \tag{7.40}$$

- Now, the expression for $\boldsymbol{Q}^\top\tilde{\boldsymbol{\omega}}$ in equation 7.13 for L^2 regularization can be rearranged as

$$\boldsymbol{Q}^\top \tilde{\boldsymbol{w}} = (\boldsymbol{\Lambda} + \alpha\boldsymbol{I})^{-1}\boldsymbol{\Lambda}\boldsymbol{Q}^\top \boldsymbol{w}^*, \tag{7.41}$$

$$\boldsymbol{Q}^\top \tilde{\boldsymbol{w}} = [\boldsymbol{I} - (\boldsymbol{\Lambda} + \alpha\boldsymbol{I})^{-1}\alpha]\boldsymbol{Q}^\top \boldsymbol{w}^*. \tag{7.42}$$

- Comparing equation 7.40 and equation 7.42, we see that if the hyperparameters ϵ, α, and τ are chosen such that

$$(\boldsymbol{I} - \epsilon\boldsymbol{\Lambda})^\tau = (\boldsymbol{\Lambda} + \alpha\boldsymbol{I})^{-1}\alpha, \tag{7.43}$$

then L^2 regularization and early stopping can be seen as equivalent (at least under the quadratic approximation of the objective function).

Early stopping

- Going even further, by taking logarithms and using the series expansion for $\log(1+x)$, we can conclude that if all λ_i are small (that is, $\epsilon\lambda_i \ll 1$ and $\lambda_i/\alpha \ll 1$) then

$$\tau \approx \frac{1}{\epsilon\alpha}, \tag{7.44}$$

$$\alpha \approx \frac{1}{\tau\epsilon}. \tag{7.45}$$

- That is, under these assumptions, the number of training iterations τ plays a role inversely proportional to the L^2 regularization parameter, and the inverse of $\tau\epsilon$ plays the role of the weight decay coefficient.

Early stopping

- Early stopping is more than the mere restriction of the trajectory length; instead, early stopping typically involves monitoring the validation set error in order to stop the trajectory at a particularly good point in space.

- Early stopping therefore has the advantage over weight decay in that it automatically determines the correct amount of regularization while weight decay requires many training experiments with different values of its hyperparameter.

Bagging and other ensemble methods

- Bagging (short for bootstrap aggregating) is a technique for reducing generalization error by combining several models.

- The idea is to train several different models separately, then have all the models vote on the output for test examples. This is an example of a general strategy in machine learning called model averaging. Techniques employing this strategy are known as ensemble methods.

- The reason that model averaging works is that different models will usually not make all the same errors on the test set.

Bagging and other ensemble methods

- Consider for example a set of k regression models. Suppose that each model makes an error ϵ_i on each example, with the errors drawn from a zero-mean multivariate normal distribution with variances $\mathbb{E}[\epsilon_i^2] = v$ and covariances $\mathbb{E}[\epsilon_i \epsilon_j] = c$.

- Then the error made by the average prediction of all the ensemble models is

$$\mathbb{E}\left[\left(\frac{1}{k}\sum_i \epsilon_i\right)^2\right] = \frac{1}{k^2}\mathbb{E}\left[\sum_i\left(\epsilon_i^2 + \sum_{j\neq i}\epsilon_i\epsilon_j\right)\right], \tag{7.50}$$

$$= \frac{1}{k}v + \frac{k-1}{k}c. \tag{7.51}$$

Bagging and other ensemble methods

- In the case where the errors are perfectly correlated and $c = v$, the mean squared error reduces to v, so the model averaging does not help at all.

- In the case where the errors are perfectly uncorrelated and $c = 0$, the expected squared error of the ensemble is only $\frac{1}{k}v$. This means that the expected squared error of the ensemble decreases linearly with the ensemble size.

- In other words, on average, the ensemble will perform at least as well as any of its members, and if the members make independent errors, the ensemble will perform significantly better than its members.

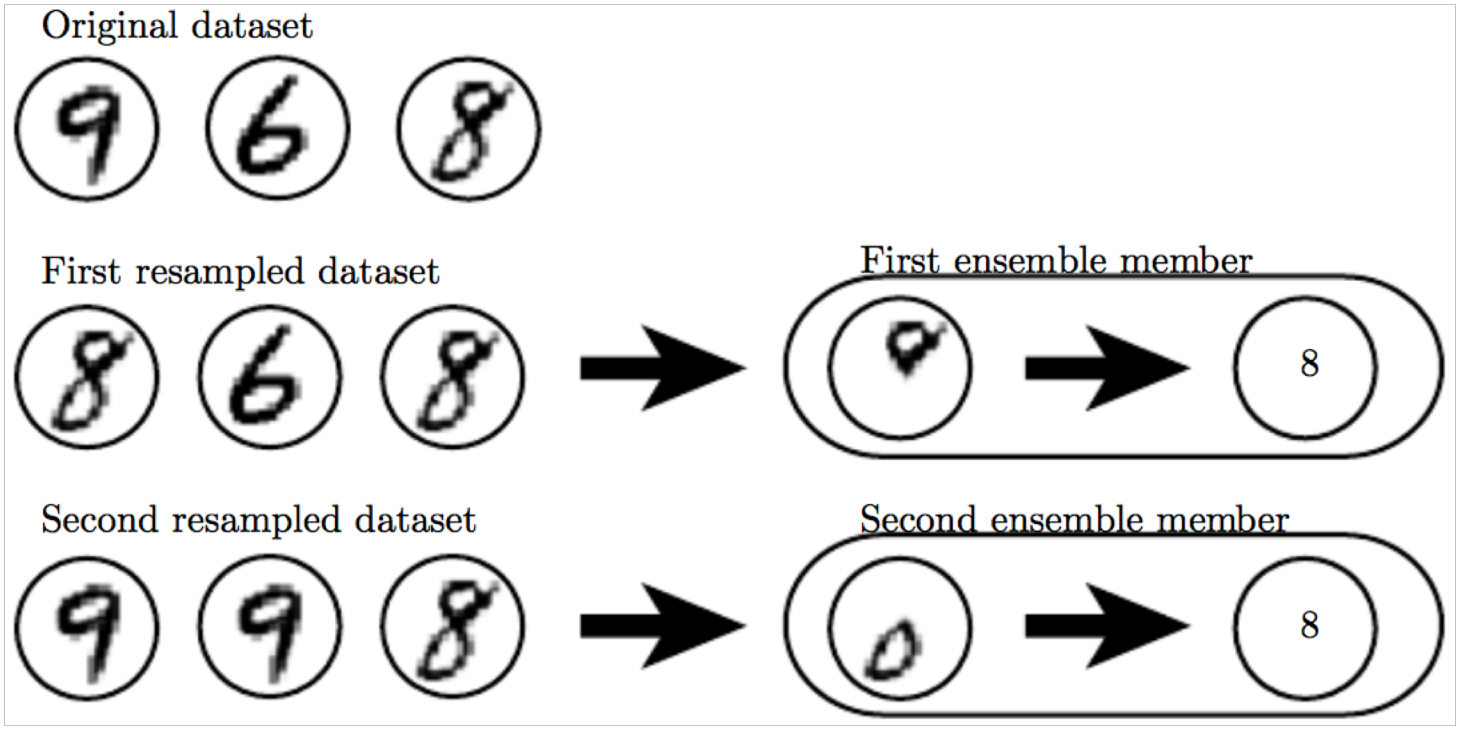
Bagging and other ensemble methods

- Bagging is a method that allows the same kind of model, training algorithm and objective function to be reused several times.

- Bagging involves constructing k different datasets. Each dataset has the same number of examples as the original dataset, but each dataset is constructed by sampling with replacement from the original dataset. This means that, with high probability, each dataset is missing some of the examples from the original dataset and contains several duplicate examples.

- Model i is then trained on dataset i. The differences between which examples are included in each dataset result in differences between the trained models.

Bagging and other ensemble methods



Bagging and other ensemble methods

- Neural networks reach a wide enough variety of solution points that they can often benefit from model averaging even if all the models are trained on the same dataset. Differences in random initialization, in random selection of minibatches, in hyperparameters, or in outcomes of nondeterministic implementations of neural networks are often enough to cause different members of the ensemble to make partially independent errors.

- Model averaging is an extremely powerful and reliable method for reducing generalization error. Its use is usually discouraged when benchmarking algorithms for scientific papers, because any machine learning algorithm can benefit substantially from model averaging at the price of increased computation and memory.

Bagging and other ensemble methods

- Machine learning contests are usually won by methods using model averaging over dozens of models.

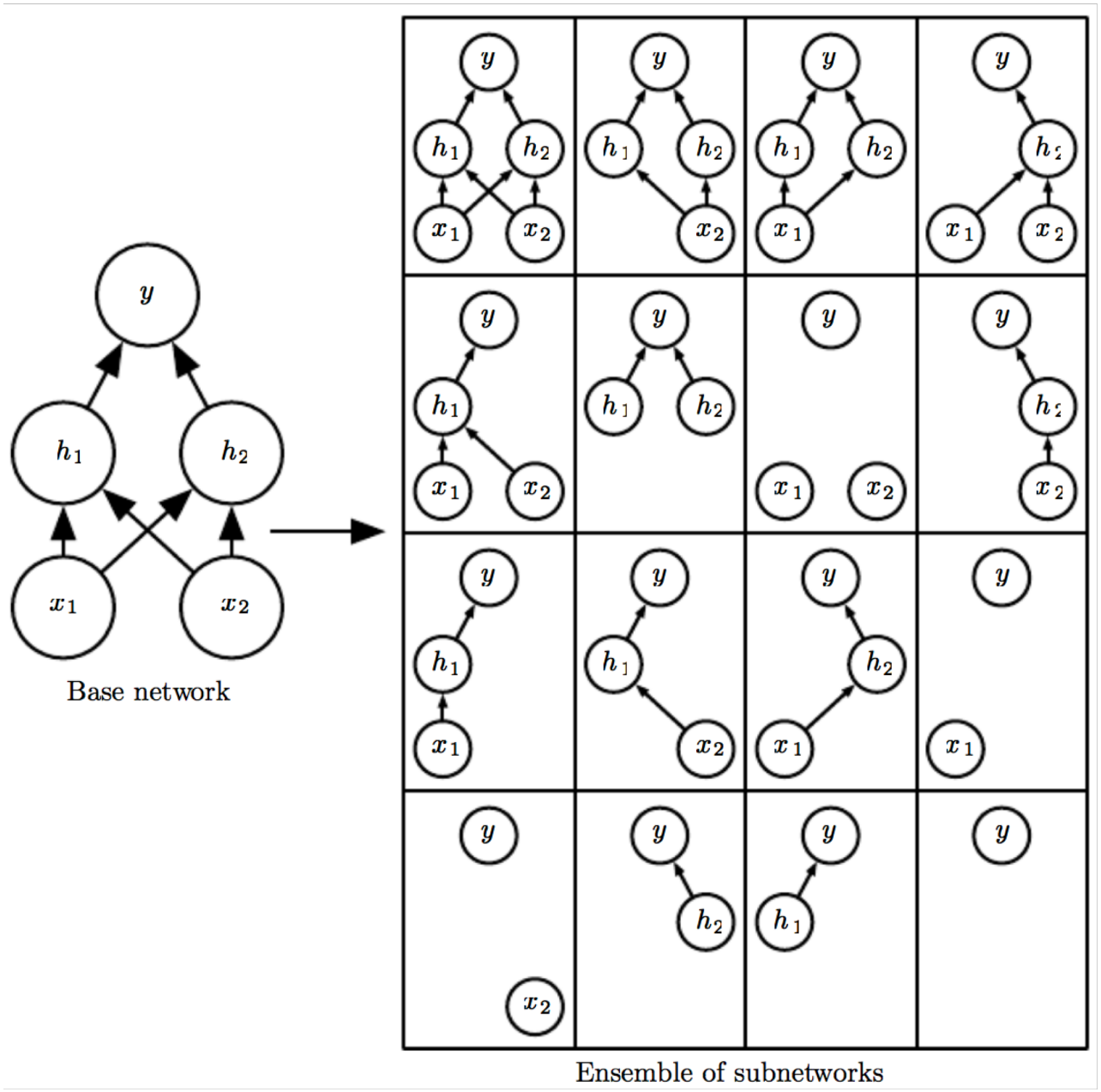
- Not all techniques for constructing ensembles are designed to make the ensemble more regularized than the individual models. For example, a technique called boosting constructs an ensemble with higher capacity than the individual models.

- Boosting has been applied to build ensembles of neural networks by incrementally adding neural networks to the ensemble. Boosting has also been applied interpreting an individual neural network as an ensemble, incrementally adding hidden units to the network.

Dropout

- Applying bagging seems impractical when each model is a large neural network, since training and evaluating such networks is costly in terms of runtime and memory.
- Dropout provides an inexpensive approximation to training and evaluating a bagged ensemble of exponentially many neural networks.
- Dropout trains the ensemble consisting of all subnetworks that can be formed by removing nonoutput units from an underlying base network.
- In most neural networks, we can effectively remove a unit from a network by multiplying its output value by zero.

Dropout



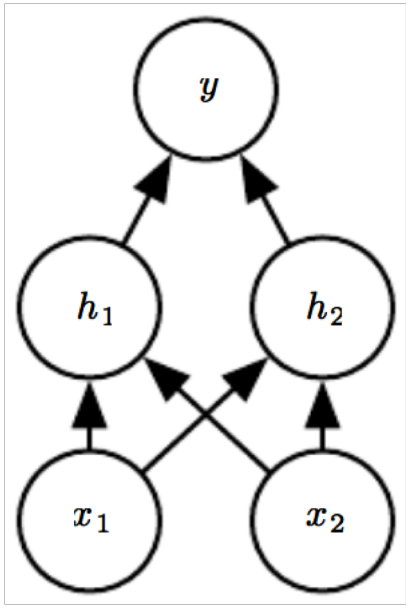
Dropout

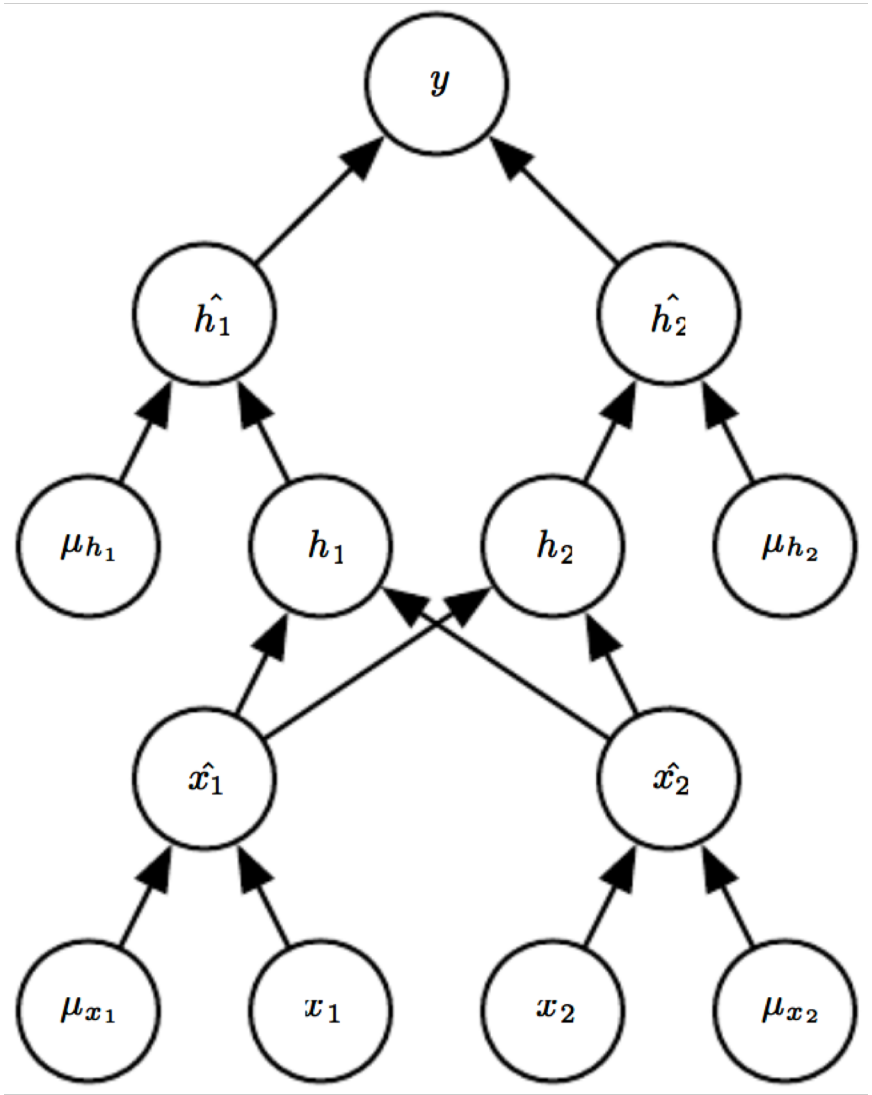
- Each time we load an example into a minibatch, we randomly sample a different binary mask to apply to all the input and hidden units in the network.

- The mask for each unit is sampled independently from all the others. The probability of sampling a mask value of one (causing a unit to be included) is a hyperparameter fixed before training begins. Typically, an input unit is included with probability 0.8, and a hidden unit is included with probability 0.5.

- We then run forward propagation, back-propagation, and the learning update as usual.

Dropout


y
h_1
h_2
x_1
x_2


y
$\hat{h_1}$
$\hat{h_2}$
μ_{h_1}
h_1
h_2
μ_{h_2}
$\hat{x_1}$
$\hat{x_2}$
μ_{x_1}
x_1
x_2
μ_{x_2}

Dropout

- More formally, suppose that a mask vector $\boldsymbol{\mu}$ specifies which units to include, and $J(\boldsymbol{\theta}, \boldsymbol{\mu})$ defines the cost of the model defined by parameters $\boldsymbol{\theta}$ and mask $\boldsymbol{\mu}$.

- Dropout training consists of minimizing $\mathbb{E}_{\boldsymbol{\mu}} J(\boldsymbol{\theta}, \boldsymbol{\mu})$. The expectation contains exponentially many terms, but we can obtain an unbiased estimate of its gradient by sampling values of $\boldsymbol{\mu}$.

- In bagging, the models are all independent. In dropout, the models share parameters, with each model inheriting a different subset of parameters from the parent neural network. This parameter sharing makes it possible to represent an exponential number of models with a tractable amount of memory.

Dropout

- In the case of bagging, each model is trained to convergence on its respective training set. In the case of dropout, typically most models are not explicitly trained at all—usually, the model is large enough that it would be infeasible to sample all possible subnetworks within the lifetime of the universe. Instead, a tiny fraction of the possible subnetworks are each trained for a single step, and the parameter sharing causes the remaining subnetworks to arrive at good settings of the parameters.

- Other than these, dropout follows the bagging algorithm.

Dropout

- To make a prediction, a bagged ensemble must accumulate votes from all its members. We refer to this process as inference.

- We assume that the model's role is to output a probability distribution. In the case of bagging, each model i produces a probability distribution $p^{(i)}(y|\boldsymbol{x})$.

- The prediction of the ensemble is given by the arithmetic mean of all these distributions,

$$\frac{1}{k}\sum_{i=1}^{k} p^{(i)}(y \mid \boldsymbol{x}). \tag{7.52}$$

Dropout

- In the case of dropout, each submodel defined by mask vector $\boldsymbol{\mu}$ defines a probability distribution $p(y|\, \boldsymbol{x}, \boldsymbol{\mu})$. The arithmetic mean over all masks is given by

$$\sum_{\boldsymbol{\mu}} p(\boldsymbol{\mu}) p(y \mid \boldsymbol{x}, \boldsymbol{\mu}), \tag{7.53}$$

where $p(\boldsymbol{\mu})$ is the probability distribution that was used to sample $\boldsymbol{\mu}$ at training time.

- Because this sum includes an exponential number of terms, it is intractable to evaluate except when the structure of the model permits some form of simplification. But so far, deep neural nets are not known to permit any tractable simplification.

Dropout

- We can approximate the inference with sampling, by averaging together the output from many masks. Even 10 to 20 masks are often sufficient to obtain good performance.

- There is an even better approach which allows us to obtain a good approximation to the predictions of the entire ensemble, at the cost of only one forward propagation.

Dropout

- The unnormalized probability distribution defined directly by the geometric mean is given by

$$\tilde{p}_{\text{ensemble}}(y \mid \boldsymbol{x}) = \sqrt[2^d]{\prod_{\boldsymbol{\mu}} p(y \mid \boldsymbol{x}, \boldsymbol{\mu})}, \tag{7.54}$$

where d is the number of units that may be dropped. Here we use a uniform distribution over $\boldsymbol{\mu}$ to simplify the presentation, but nonuniform distributions are also possible.

- To make predictions we must renormalize the ensemble:

$$p_{\text{ensemble}}(y \mid \boldsymbol{x}) = \frac{\tilde{p}_{\text{ensemble}}(y \mid \boldsymbol{x})}{\sum_{y'} \tilde{p}_{\text{ensemble}}(y' \mid \boldsymbol{x})}. \tag{7.55}$$

Dropout

- A key insight involved in dropout is that we can approximate p_{ensemble} by evaluating $p(y|\boldsymbol{x})$ in one model: the model with all units, but with the weights going out of unit i multiplied by the probability of including unit i.

- The motivation for this modification is to capture the right expected value of the output from that unit. We call this approach the weight scaling inference rule.

- There is not yet any theoretical argument for the accuracy of this approximate inference rule in deep nonlinear networks, but empirically it performs very well.

Dropout

- Because we usually use an inclusion probability of $\frac{1}{2}$, the weight scaling rule usually amounts to dividing the weights by 2 at the end of training, and then using the model as usual.

- Another way to achieve the same result is to multiply the states of the units by 2 during training.

- Either way, the goal is to make sure that the expected total input to a unit at test time is roughly the same as the expected total input to that unit at train time, even though half the units at train time are missing on average.

Dropout

- For many classes of models that do not have nonlinear hidden units, the weight scaling inference rule is exact. For a simple example, consider a $\mathrm{softmax}$ regression classifier with n input variables represented by the vector $\mathbf{v}$:

$$P(\mathrm{y} = y \mid \mathbf{v}) = \mathrm{softmax}\left(\boldsymbol{W}^\top \mathbf{v} + \boldsymbol{b}\right)_y . \tag{7.56}$$

- We can index into the family of submodels by element-wise multiplication of the input with a binary vector d:

$$P(\mathrm{y} = y \mid \mathbf{v}; \boldsymbol{d}) = \mathrm{softmax}\left(\boldsymbol{W}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)_y . \tag{7.57}$$

Dropout

- The ensemble predictor is defined by renormalizing the geometric mean over all ensemble members' predictions:

$$P_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) = \frac{\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v})}{\sum_{y'} \tilde{P}_{\text{ensemble}}(\mathrm{y} = y' \mid \mathbf{v})}, \tag{7.58}$$

where

$$\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) = \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} P(\mathrm{y} = y \mid \mathbf{v}; \boldsymbol{d})}. \tag{7.59}$$

Dropout

- To see that the weight scaling rule is exact, we can simplify $\tilde{P}_{\text{ensemble}}$

$$\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) = \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} P(\mathrm{y} = y \mid \mathbf{v}; \boldsymbol{d})} \tag{7.60}$$

$$= \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \text{softmax}\left(\boldsymbol{W}^\top (\boldsymbol{d} \odot \mathbf{v}) + \boldsymbol{b}\right)_y} \tag{7.61}$$

$$= \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \frac{\exp\left(\boldsymbol{W}_{y,:}^\top (\boldsymbol{d} \odot \mathbf{v}) + b_y\right)}{\sum_{y'} \exp\left(\boldsymbol{W}_{y',:}^\top (\boldsymbol{d} \odot \mathbf{v}) + b_{y'}\right)}} \tag{7.62}$$

$$= \frac{\sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \exp\left(\boldsymbol{W}_{y,:}^\top (\boldsymbol{d} \odot \mathbf{v}) + b_y\right)}}{\sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \sum_{y'} \exp\left(\boldsymbol{W}_{y',:}^\top (\boldsymbol{d} \odot \mathbf{v}) + b_{y'}\right)}} \tag{7.63}$$

Dropout

- Because $\tilde{P}$ will be normalized, we can safely ignore multiplication by factors that are constant with respect to y:

$$\tilde{P}_{\text{ensemble}}(\mathrm{y} = y \mid \mathbf{v}) \propto \sqrt[2^n]{\prod_{\boldsymbol{d} \in \{0,1\}^n} \exp\left(\boldsymbol{W}_{y,:}^{\top}(\boldsymbol{d} \odot \mathbf{v}) + b_y\right)} \tag{7.64}$$

$$= \exp\left(\frac{1}{2^n} \sum_{\boldsymbol{d} \in \{0,1\}^n} \boldsymbol{W}_{y,:}^{\top}(\boldsymbol{d} \odot \mathbf{v}) + b_y\right) \tag{7.65}$$

$$= \exp\left(\frac{1}{2} \boldsymbol{W}_{y,:}^{\top} \mathbf{v} + b_y\right). \tag{7.66}$$

- Substituting this back into equation 7.58, we obtain a softmax classifier with weights $\frac{1}{2}\boldsymbol{W}$.

Dropout

- The weight scaling rule is also exact in other settings, including regression networks with conditionally normal outputs as well as deep networks that have hidden layers without nonlinearities.

- However, the weight scaling rule is only an approximation for deep models that have nonlinearities. Though the approximation has not been theoretically characterized, it often works well, empirically.

Dropout

- It was found experimentally that the weight scaling approximation can work better than Monte Carlo approximations to the ensemble predictor. This held true even when the Monte Carlo approximation was allowed to sample up to 1,000 subnetworks.

- It was found that some models obtain better classification accuracy using twenty samples and the Monte Carlo approximation. It appears that the optimal choice of inference approximation is problem dependent.

- It was shown that dropout is more effective than other standard computationally inexpensive regularizers, such as weight decay.

Dropout

- One advantage of dropout is that it is very computationally cheap.
- Using dropout during training requires only $O(n)$ computation per example per update, to generate n random binary numbers and multiply them by the state. Depending on the implementation, it may also require $O(n)$ memory to store these binary numbers until the back-propagation stage.
- Running inference in the trained model has the same cost per example as if dropout were not used, though we must pay the cost of dividing the weights by 2 once before beginning to run inference on examples.

Dropout

- Another significant advantage of dropout is that it does not significantly limit the type of model or training procedure that can be used. It works well with nearly any model that uses a distributed representation and can be trained with stochastic gradient descent. This includes feedforward neural networks, probabilistic models such as restricted Boltzmann machines, and recurrent neural networks.

- Many other regularization strategies of comparable power impose more severe restrictions on the architecture of the model.

Dropout

- Because dropout is a regularization technique, it reduces the effective capacity of a model. To offset this effect, we must increase the size of the model. Typically the optimal validation set error is much lower when using dropout, but this comes at the cost of a much larger model and many more iterations of the training algorithm.

- For very large datasets, regularization confers little reduction in generalization error. The computational cost of using dropout and larger models may outweigh the benefit of regularization.

- When extremely few labeled training examples are available, dropout is less effective. When additional unlabeled data is available, unsupervised feature learning can gain an advantage over dropout.

Dropout

- Another view of dropout: Dropout trains not just a bagged ensemble of models, but an ensemble of models that share hidden units.

- This means each hidden unit must be able to perform well regardless of which other hidden units are in the model. Hidden units must be prepared to be swapped and interchanged between models.

- Dropout thus regularizes each hidden unit to be not merely a good feature but a feature that is good in many contexts.

Dropout

- A large portion of the power of dropout arises from the fact that the masking noise is applied to the hidden units.
- This can be seen as a form of highly intelligent, adaptive destruction of the information content of the input rather than destruction of the raw values of the input.
- For example, if the model learns a hidden unit h_i that detects a face by finding the nose, then dropping h_i corresponds to erasing the information that there is a nose in the image. The model must learn another h_i, that either redundantly encodes the presence of a nose or detects the face by another feature, such as the mouth.

Dropout

- Traditional noise injection techniques that add unstructured noise at the input are not able to randomly erase the information about a nose from an image of a face unless the magnitude of the noise is so great that nearly all the information in the image is removed.

- Destroying extracted features rather than original values allows the destruction process to make use of all the knowledge about the input distribution that the model has acquired so far.

Dropout

- Another important aspect of dropout is that the noise is multiplicative. If the noise were additive with fixed scale, then a rectified linear hidden unit h_i with added noise ϵ could simply learn to have h_i become very large in order to make the added noise ϵ insignificant by comparison.

- Another deep learning algorithm, batch normalization, reparametrizes the model in a way that introduces both additive and multiplicative noise on the hidden units at training time. The primary purpose of batch normalization is to improve optimization, but the noise can have a regularizing effect, and sometimes makes dropout unnecessary.