Deep learning: Different forms of Regularization

28/07/2025

1 Taylor series

The Taylor series of a real or complex-valued function $f(x)$ that is infinitely differentiable at a real or complex number a is the power series

$$f(a)+\frac{f'(a)}{1!}(x-a)+\frac{f''(a)}{2!}(x-a)^2+\frac{f'''(a)}{3!}(x-a)^3+\cdots,$$

where n ! denotes the factorial of n. In the more compact sigma notation, this can be written as

$$\sum_{n=0}^{\infty}\frac{f^{(n)}(a)}{n!}(x-a)^n,$$

where $f^{(n)}(a)$ denotes the nth derivative of f evaluated at the point a.

For a function $f(x,y)$ that depends on two variables, x and y, the Taylor series to second order about the point (a,b) is

$$f(a,b)+(x-a)f_x(a,b)+(y-b)f_y(a,b)+\frac{1}{2!}\left((x-a)^2f_{xx}(a,b)+2(x-a)(y-b)f_{xy}(a,b)+(y-b)^2f_{yy}(a,b)\right)$$

where the subscripts denote the respective partial derivatives. A second-order Taylor series expansion (or quadratic approximation) of a scalar-valued function of more than one variable can be written compactly as

$$T(\mathbf{x})=f(\mathbf{a})+(\mathbf{x}-\mathbf{a})^\top\nabla f(\mathbf{a})+\frac{1}{2!}(\mathbf{x}-\mathbf{a})^\top\left\{\nabla^2 f(\mathbf{a})\right\}(\mathbf{x}-\mathbf{a})$$

where $\nabla f(\mathbf{a})$ is the gradient of f evaluated at $\mathbf{x}=\mathbf{a}$ and $\nabla^2 f(\mathbf{a})$ is the Hessian matrix.

In order to compute a Taylor series expansion around point $(a,b)=(0,0)$ of the function

$$f(x,y)=e^x\ln(1+y),$$

one first computes all the necessary partial derivatives: $f_x=e^x\ln(1+y)$
$f_y=\frac{e^x}{1+y}$
$f_{xx}=e^x\ln(1+y)$

$f_{yy} = -\frac{e^x}{(1+y)^2}$
$f_{xy} = f_{yx} = \frac{e^x}{1+y}$.

Evaluating these derivatives at the origin gives the Taylor coefficients $f_x(0,0) =$
0
$f_y(0,0) = 1$
$f_{xx}(0,0) = 0$
$f_{yy}(0,0) = -1$
$f_{xy}(0,0) = f_{yx}(0,0) = 1$

Substituting these values in to the general formula
$$T(x,y) = f(a,b) + (x-a)f_x(a,b) + (y-b)f_y(a,b) \quad + \frac{1}{2!}\left((x-a)^2 f_{xx}(a,b) + 2(x-a)(y-b)f_{xy}(a,b) + (y-b)^2 f_{yy}(a,b)\right) + \cdots$$

produces

$$\begin{aligned} T(x,y) &= 0 + 0(x-0) + 1(y-0) + \frac{1}{2}\left(0(x-0)^2 + 2(x-0)(y-0) + (-1)(y-0)^2\right) + \cdots \\ &= y + xy - \frac{y^2}{2} + \cdots \end{aligned}$$

Since $\ln(1+y)$ is analytic in $|y| < 1$, we have $e^x \ln(1+y) = y + xy - \frac{y^2}{2} + \cdots, \quad |y| < 1$

Suppose $f : R^n \to R$ is a function taking as input a vector $\mathbf{x} \in R^n$ and outputting a scalar $f(\mathbf{x}) \in R$. If all second-order partial derivatives of f exist, then the Hessian matrix $\mathbf{H}$ of f is a square $n \times n$ matrix, usually defined and arranged as

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

2 $\mathbf{L}^2$*regularization*

Regularization has been used for decades prior to the advent of deep learning. Linear models such as linear regression and logistic regression allow simple, straightforward, and effective regularization strategies.

Many regularization approaches are based on limiting the capacity of models, such as neural networks, linear regression, or logistic regression, by adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the objective function J. We denote the regularized objective function by $\tilde{J}$:

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

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.

Here what we try to achieve by regularization is to reduction of testing error by making the models less complex so that they can generalize well.

When our training algorithm minimizes the regularized objective function $\tilde{J}$ it will decrease both the original objective J on the training data and some measure of the size of the parameters $\boldsymbol{\theta}$ (or some subset of the parameters). Different choices for the parameter norm Ω can result in different solutions being preferred. In this section, we discuss the effects of the various norms when used as penalties on the model parameters.

One of the simplest and most common kinds of parameter norm penalty: the L^2 parameter norm penalty commonly known as weight decay . This regularization strategy drives the weights closer to the origin by adding a regularization term $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2$ 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. To simplify the presentation, we assume no bias parameter, so $\boldsymbol{\theta}$ is just $\boldsymbol{w}$. Such a model has the following total objective function:

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

with the corresponding parameter gradient

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

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

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

Written another way, the update is

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

We can see that 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, just before performing the usual gradient update. This describes what happens in a single step. But what happens over the entire course of training?

We will 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, $\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} J(\boldsymbol{w})$. 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. The approximation $\hat{J}$ is given by

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

where $\boldsymbol{H}$ is the Hessian matrix of J with respect to $\boldsymbol{w}$ evaluated at $\boldsymbol{w}^*$. There is no first-order term in this quadratic approximation, because $\boldsymbol{w}^*$ is defined to be a minimum, where the gradient vanishes. Likewise, because $\boldsymbol{w}^*$ is the location of a minimum of J, we can conclude that $\boldsymbol{H}$ is positive semidefinite. The minimum of J occurs where its gradient

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

is equal to $\mathbf{0}$.

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

$$\begin{aligned} \alpha\tilde{\boldsymbol{w}} + \boldsymbol{H}(\tilde{\boldsymbol{w}} - \boldsymbol{w}^*) &= 0 \\ (\boldsymbol{H} + \alpha\boldsymbol{I})\tilde{\boldsymbol{w}} &= \boldsymbol{H}\boldsymbol{w}^* \\ \tilde{\boldsymbol{w}} &= (\boldsymbol{H} + \alpha\boldsymbol{I})^{-1}\boldsymbol{H}\boldsymbol{w}^* \end{aligned}$$

As α approaches 0 , the regularized solution $\tilde{w}$ approaches $\boldsymbol{w}^*$. But what happens as α grows? 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 $\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$.

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

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

Along the directions where the eigenvalues of $\boldsymbol{H}$ 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.

3 L^1*regularization*

While L^2 weight decay is the most common form of weight decay, there are other ways to penalize the size of the model parameters. Another option is to use L^1 regularization.

Formally, L^1 regularization on the model parameter $\boldsymbol{w}$ is defined as

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

that is, as the sum of absolute values of the individual parameters. The regularized objective function $\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$ is given by

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

A Similar discussion as in the above case can be carried for looking at what is really happening when we apply L1 regularization.

4 Dataset augmentation

The best way to make a machine learning model generalize better is to train it on more data. Of course, in practice, the amount of data we have is limited. One way to get around this problem is to create fake data and add it to the training set. For some machine learning tasks, it is reasonably straightforward to create new fake data.

This approach is easiest for classification. A classifier needs to take a complicated, high-dimensional input $\boldsymbol{x}$ and summarize it with a single category identity y. This means that 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 has been a particularly effective technique for a specific classification problem: object recognition. Images are high dimensional and include an enormous range of factors of variation, many of which can be easily simulated. 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$\circ$ rotations are not appropriate ways of augmenting datasets for these tasks.

5 Semi-supervised learning

Supervised learning algorithms deal with labelled data. And unsupervised learning algorithms deal with unlabelled data. Semi-supervised learning is a branch of machine learning that combines a small amount of labeled data with a large amount of unlabeled data during training. Semi supervised learning is a form in which we try to club both labelled and unlabelled data to create models.

So suppose you have a bunch of images of dogs and cats and you need to classify those images. But you just have some images which are labelled(that means you know whether it is a dog or a cat) and certain images that are unlabelled. So your labeled data is like you know the input x(an image of a dog say) and its label y(which is dog). And an unlabelled data is like an image x you have with you but no information about y.

In such cases one can use a neural network to initially train the model with available labelled data and then after that send in the unlabelled data (x's) and predict the corresponding labels. Thus finally one can fit a model with all these available new labelled data. This kind of technique is called semi supervised learning.

Another example is like look how google photos work. already the algorithm makes clusters of images using the unsupervised learning techniques and from the limited labels it can generate using human help it classifies the images in to the actual classes.

6 Ensemble methods

The one kind of ensemble learning method we have seen is bagging (or bootstrap aggregation) in the case of decision trees. The same can be used in other scenarios also.

Bagging (short for bootstrap aggregating) is a technique for reducing generalization error by combining several models (Breiman, 1994). 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.

7 Dropouts

Note that::

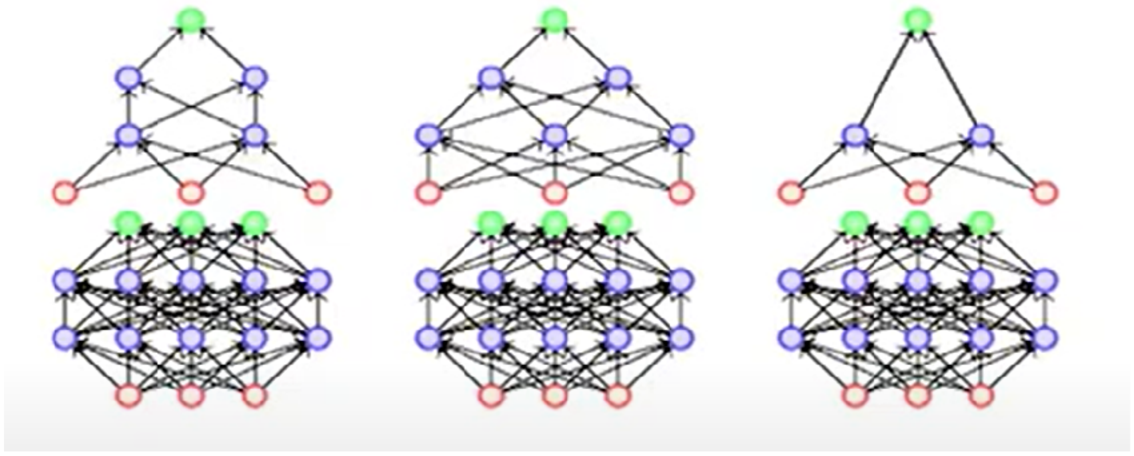
- Typically model averaging(bagging ensemble) always helps

- Training several large neural networks for making an ensemble is prohibitively expensive

- Option 1: Train several neural networks having different architectures(obviously expensive)

- Option 2: Train multiple instances of the same network using different training samples (again expensive)

- Even if we manage to train with option 1 or option 2, combining several models at test time is infeasible in real time applications.



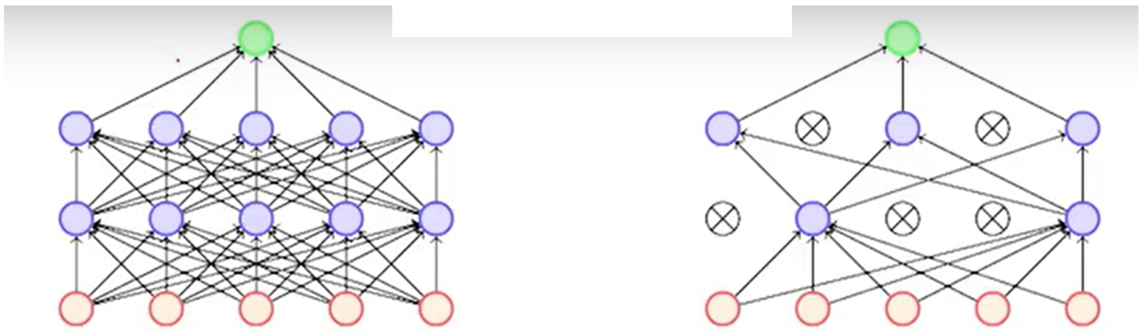
- Dropout is a technique which addresses both these issues.

- Effectively it allows training several neural networks without any significant computational overhead.

- Dropout refers to dropping out units

- Temporarily remove a node and all its incoming/outgoing connections resulting in a thinned network

- Each node is retained with a fixed probability (typically $p = 0.5$) for hidden nodes and $p = 0.8$ for visible nodes



- Suppose a neural network has n nodes

- Using the dropout idea, each node can be retained or dropped

- For example, in the above case we drop 5 nodes to get a thinned network

- Given a total of nodes, what are the total number of thinned networks that can be formed? 2^n

- Of course, this is prohibitively large and we cannot possibly train so many networks

- We initialize all the parameters (weights) of the network and start training

- For the first training instance (or mini-batch), we apply dropout resulting in the thinned network

- We compute the loss and backpropagate

- Which parameters will we update? Only those which are active

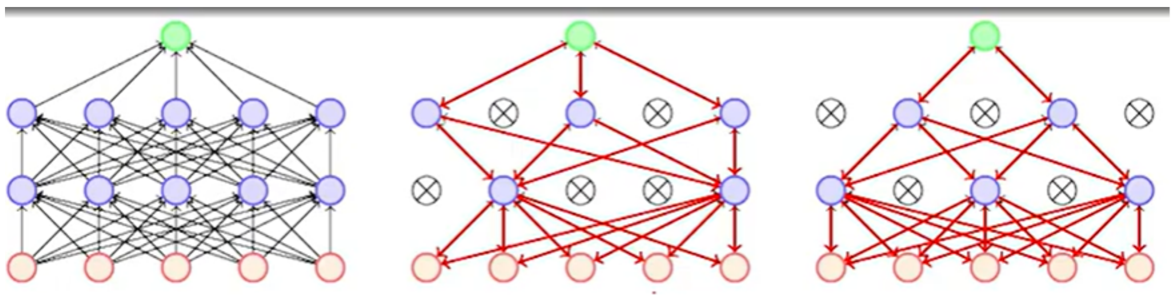
- For the second training instance, we again apply dropout resulting in a different thinned network

- We again compute the loss and backpropagate to the active weights

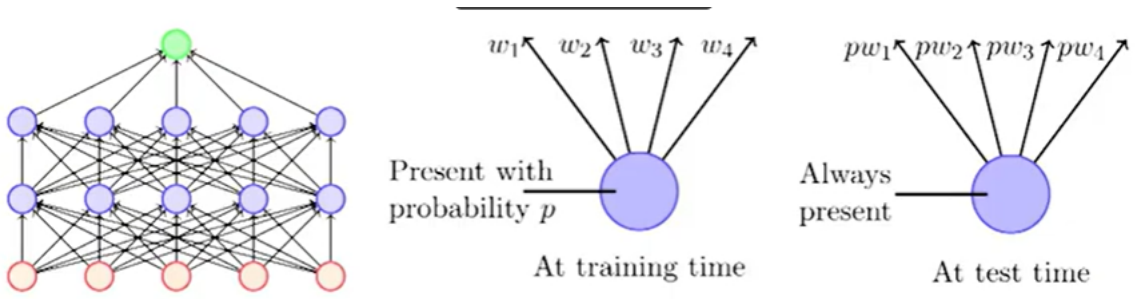
- If the weight was active for both the training instances then it would have received two updates by now

- If the weight was active for only one of the training instances then it would have received only one updates by now

- Each thinned network gets trained rarely (or even never) but the parameter sharing ensures that no model has untrained or poorly trained parameters

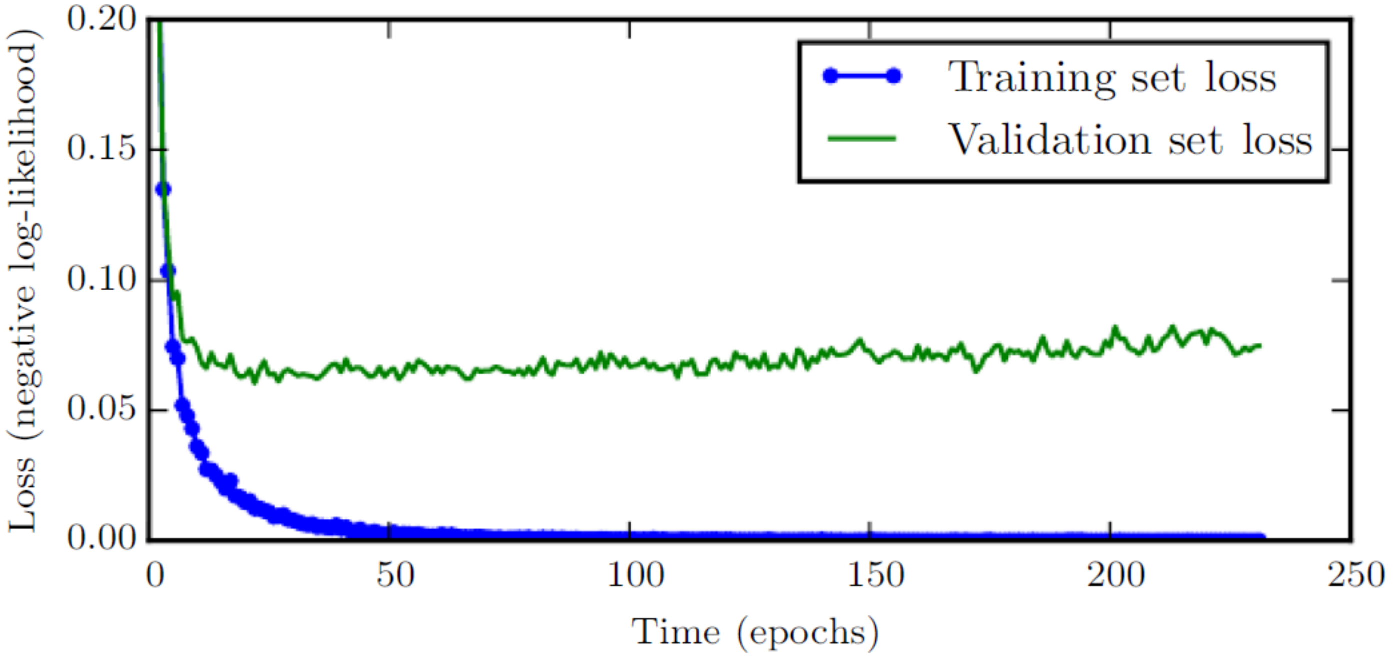


- What happens at test time?
- Impossible to aggregate the outputs of 2^n thinned networks
- Instead we use the full Neural Network and scale the output of each node by the fraction of times it was on during training



Advantages
- Prevents hidden units from adapting
- Essentially a hidden unit cannot rely too much on other units as they may get dropped out any time
- Each hidden unit has to learn to be more robust to these random dropouts

8 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. See above figure for an example of this behavior, which occurs reliably. 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 some pre-specified number of iterations. 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.

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

Algorithm 7.1 The early stopping meta-algorithm for determining the best amount of time to train. This meta-algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set. **Algorithm**

```
Let $n$ be the number of steps between evaluations.
Let $p$ be the "patience," the number of times to observe worsening validation set
error before giving up.
Let $\boldsymbol{\theta}_o$ be the initial parameters.
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_o$
$i \leftarrow 0$
$j \leftarrow 0$
$v \leftarrow \infty$
$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
$i^* \leftarrow i$
while $j < p$ do
  Update $\boldsymbol{\theta}$ by running the training algorithm for $n$ steps.
  $i \leftarrow i + n$
  $v' \leftarrow \text{ValidationSetError}(\boldsymbol{\theta})$
  if $v' < v$ then
    $j \leftarrow 0$
    $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
    $i^* \leftarrow i$
    $v \leftarrow v'$
  else
    $j \leftarrow j + 1$
  end if
end while
Best parameters are $\boldsymbol{\theta}^*$, best number of training steps is $i^*$.
```

Now when we consider early stopping method we need a validation set as well. Usually when we fit a model we split the data to training and testing set, use the training data to create a model and then use the testing data to evaluate the model. But our early stopping method requires a training set, testing set and a validation set. Here validation set is used as a part of training to find the optimal model parameters and hyper parameters.

So, 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.

We can also show that in the case of a simple linear model with a quadratic error function and simple gradient descent—early stopping is equivalent to L2 regularization. To compare with classical L^2 regularization, we examine a simple setting where the only parameters are linear weights ($\boldsymbol{\theta} = \boldsymbol{w}$). We can model the cost function J with a quadratic approximation in the neighborhood of the empirically optimal value of the weights $\boldsymbol{w}^*$:

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

where $\boldsymbol{H}$ is the Hessian matrix of J with respect to $\boldsymbol{w}$ evaluated at $\boldsymbol{w}^*$. Given the assumption that $\boldsymbol{w}^*$ is a minimum of $J(\boldsymbol{w})$, we know that $\boldsymbol{H}$ is Under a local Taylor series approximation, the gradient is given by

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

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, [3] $\boldsymbol{w}^{(0)} = 0$. Let us study the approximate behavior of gradient descent on J by analyzing gradient descent on $\hat{J}$:

$$\begin{aligned}
\boldsymbol{w}^{(\tau)} &= \boldsymbol{w}^{(\tau-1)} - \epsilon \nabla_{\boldsymbol{w}} \hat{J}\left(\boldsymbol{w}^{(\tau-1)}\right) \\
&= \boldsymbol{w}^{(\tau-1)} - \epsilon \boldsymbol{H}\left(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*\right) \\
\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^* &= (\boldsymbol{I} - \epsilon \boldsymbol{H})\left(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*\right)
\end{aligned}$$

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

$$\begin{aligned}
\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^* &= \left(\boldsymbol{I} - \epsilon \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top\right)\left(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*\right) \\
\boldsymbol{Q}^\top\left(\boldsymbol{w}^{(\tau)} - \boldsymbol{w}^*\right) &= (\boldsymbol{I} - \epsilon \boldsymbol{\Lambda})\boldsymbol{Q}^\top\left(\boldsymbol{w}^{(\tau-1)} - \boldsymbol{w}^*\right)
\end{aligned}$$

Assuming that $\boldsymbol{w}^{(0)} = 0$ and 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:

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

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

$$\begin{aligned}
\boldsymbol{Q}^\top \tilde{\boldsymbol{w}} &= (\boldsymbol{\Lambda} + \alpha \boldsymbol{I})^{-1}\boldsymbol{\Lambda}\boldsymbol{Q}^\top \boldsymbol{w}^*, \\
\boldsymbol{Q}^\top \tilde{\boldsymbol{w}} &= \left[\boldsymbol{I} - (\boldsymbol{\Lambda} + \alpha \boldsymbol{I})^{-1}\alpha\right]\boldsymbol{Q}^\top \boldsymbol{w}^*.
\end{aligned}$$

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

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

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