Regularization for Deep Learning

B AJAY RAM
ASSISTANT PROFESSOR
CSE DEPARTMENT

INTRODUCTION

- A central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs.
- 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.
- Regularization can be defined as “any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.”
- Some put extra constraints on a machine learning model, such as adding restrictions on the parameter values. Some add extra terms in the objective function that can be thought of as corresponding to a soft constraint on the parameter values.
- If chosen carefully, these extra constraints and penalties can lead to improved performance on the test set.

- Sometimes these constraints and penalties are designed to encode specific kinds of prior knowledge.
- Other times, these constraints and penalties are designed to express a generic preference for a simpler model class in order to promote generalization.
- Sometimes penalties and constraints are necessary to make an underdetermined problem determined.
- Other forms of regularization, known as ensemble methods, combine multiple hypotheses that explain the training data.

Parameter Norm Penalties

- 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 $\Omega(\theta)$ to the objective function J.
- We denote the regularized objective function by $\tilde{J}$:

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

where $\alpha \in [0, \infty)$ is a hyper parameter 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.

- 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 θ.
- Different choices for the parameter norm Ω can result in different solutions being preferred.

L2 Parameter Regularization

- The L2 parameter norm penalty commonly known as weight decay.
- This regularization strategy drives the weights closer to the origin1 by adding a regularization term $\Omega(\theta) = 1/2||w||2\ 2$ to the objective function.
- In other academic communities, L2 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 θ is just w.
- Such a model has the following total objective function

$$J\tilde{}(w; X, y) = \alpha/2\ w\ \hat{}Tw + J(w; X, y),$$

with the corresponding parameter gradient

$$\nabla wJ\tilde{}(w; X, y) = \alpha w + \nabla wJ(w; X, y)$$

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

$$w \leftarrow w - €(\alpha w + \nabla w\ J(w; X, y)).$$

Written another way, the update is:

$$w \leftarrow (1 - €\alpha)w - €\nabla wJ(w; X, y).$$

- 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, $w* = \arg\min_w J(w)$. 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

$$\hat{J}(\theta) = J(w*) + 1/2\,(w - w*)^T H(w - w*),$$

where H is the Hessian matrix of J with respect to w evaluated at w∗. There is no first-order term in this quadratic approximation, because w∗ is defined to be a minimum, where the gradient vanishes

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

$$\nabla_w \hat{J}(w) = H(w - w*) \text{ is equal to } 0$$

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 $\tilde{w}$ to represent the location of the minimum.

$$\alpha\tilde{w} + H(\tilde{w} - w*) = 0$$

$$(H + \alpha I)\tilde{w} = Hw*$$

$$\tilde{w} = (H + \alpha I)^{-1} Hw*.$$

- As α approaches 0, the regularized solution w˜ approaches w∗. But what happens as α grows? Because H is real and symmetric, we can decompose it into a diagonal matrix Λ and an orthonormal basis of eigenvectors, Q, such that H = QΛQ^T. Applying the decomposition

$$\mathbf{w\tilde{} = (Q\Lambda Q^T + \alpha I)^{-1} Q\Lambda Q^T w*}$$
$$\mathbf{= [Q(\Lambda + \alpha I)Q^T]^{-1} Q\Lambda Q^T w*}$$
$$\mathbf{= Q(\Lambda + \alpha I)^{-1} \Lambda Q^T w*.}$$

Along the directions where the eigenvalues of H are relatively large, for example, where λi >> α, the effect of regularization is relatively small. However, components with λi <<α will be shrunk to have nearly zero magnitude.

Applying the analysis again, we will be able to obtain a special case of the same results, but with the solution now phrased in terms of the training data. For linear regression, the cost function is the sum of squared errors:

$$\mathbf{(Xw - y)^T (Xw - y).}$$

When we add L2 regularization, the objective function changes to

$$\mathbf{(Xw - y)^T (Xw - y) + 1/2\ \alpha w^T w.}$$

This changes the normal equations for the solution from

$$\mathbf{w = (X^TX)^{-1} X^Ty \text{ to } w = (X^TX + \alpha I)^{-1} X^Ty.}$$

L2 regularization causes the learning algorithm to “perceive” the input 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.

Norm Penalties as Constrained Optimization

- Consider the cost function regularized by a parameter norm penalty:

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

- we can minimize a function subject to constraints by constructing a generalized Lagrange function, consisting of the original objective function plus a set of penalties.
- Each penalty is a product between a coefficient, called a Karush–Kuhn–Tucker (KKT) multiplier, and a function representing whether the constraint is satisfied.
- If we wanted to constrain $\Omega(\theta)$ to be less than some constant k, we could construct a generalized Lagrange function

$$L(\theta, \alpha; X, y) = J(\theta; X, y) + \alpha(\Omega(\theta) - k).$$

- The solution to the constrained problem is given by

$$\theta^* = \arg\theta \min \max \alpha,\alpha \geq o\ L(\theta, \alpha)$$

- To gain some insight into the effect of the constraint, we can fix α^* and view the problem as just a function of θ:

$$\theta^* = \arg\min_\theta L(\theta, \alpha^*) = \arg\min_\theta J(\theta; X, y) + \alpha^*\Omega(\theta).$$

- This is exactly the same as the regularized training problem of minimizing $\tilde{J}$. We can thus think of a parameter norm penalty as imposing a constraint on the weights.
- If Ω is the L^2 norm, then the weights are constrained to lie in an L^2 ball. If Ω is the L^1 norm, then the weights are constrained to lie in a region of limited L^1 norm.
- Usually we do not know the size of the constraint region that we impose by using weight decay with coefficient α^* because the value of α^* does not directly tell us the value of k.
- While we do not know the exact size of the constraint region, we can control it roughly by increasing or decreasing α in order to grow or shrink the constraint region. Larger α will result in a smaller constraint region. Smaller α will result in a larger constraint region.
- Sometimes we may wish to use explicit constraints rather than penalties. We can modify algorithms such as stochastic gradient descent to take a step downhill on $J(\theta)$ and then project θ back to the nearest point that satisfies $\Omega(\theta) < k$.
- This can be useful if we have an idea of what value of k is appropriate and do not want to spend time searching for the value of α that corresponds to this k.
- Another reason to use explicit constraints and reprojection rather than enforcing constraints with penalties is that penalties can cause non-convex optimization procedures to get stuck in local minima corresponding to small θ.

- When training with a penalty on the norm of the weights, these configurations can be locally optimal, even if it is possible to significantly reduce J by making the weights larger.
- Explicit constraints implemented by re-projection can work much better in these cases because they do not encourage the weights to approach the origin. Explicit constraints implemented by re-projection only have an effect when the weights become large and attempt to leave the constraint region.
- Finally, explicit constraints with reprojection can be useful because they impose some stability on the optimization procedure.
- When using high learning rates, it is possible to encounter a positive feedback loop in which large weights induce large gradients which then induce a large update to the weights.
- If these updates consistently increase the size of the weights, then θ rapidly moves away from the origin until numerical overflow occurs.
- Explicit constraints with reprojection prevent this feedback loop from continuing to increase the magnitude of the weights without bound.
- Constraining the norm of each column separately prevents any one hidden unit from having very large weights.
- In practice, column norm limitation is always implemented as an explicit constraint with reprojection.

Regularization and Under-Constrained Problems

- In some cases, regularization is necessary for machine learning problems to be properly defined. Many linear models in machine learning, including linear regression and PCA, depend on inverting the matrix X^TX.
- This is not possible whenever X^TX is singular. This matrix can be singular whenever the data generating distribution truly has no variance in some direction, or when no variance is observed in some direction because there are fewer examples (rows of X) than input features (columns of X).
- In this case, many forms of regularization correspond to inverting X^TX + αI instead. This regularized matrix is guaranteed to be invertible.
- These linear problems have closed form solutions when the relevant matrix is invertible. It is also possible for a problem with no closed form solution to be underdetermined.
- An example is logistic regression applied to a problem where the classes are linearly separable. If a weight vector w is able to achieve perfect classification, then 2w will also achieve perfect classification and higher likelihood.

- An iterative optimization procedure like stochastic gradient descent will continually increase the magnitude of w and, in theory, will never halt.
- In practice, a numerical implementation of gradient descent will eventually reach sufficiently large weights to cause numerical overflow, at which point its behavior will depend on how the programmer has decided to handle values that are not real numbers.
- Most forms of regularization are able to guarantee the convergence of iterative methods applied to underdetermined problems. For example, weight decay will cause gradient descent to quit increasing the magnitude of the weights when the slope of the likelihood is equal to the weight decay coefficient.
- The idea of using regularization to solve underdetermined problems extends beyond machine learning. The same idea is useful for several basic linear algebra problems.
- We can thus interpret the pseudo inverse as stabilizing underdetermined problems using regularization.

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 **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 (**x**, **y**) pairs easily just by transforming the **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 variety of factors of variation, many of which can be easily simulated.

- 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 by using the convolution and pooling techniques described here. Many other operations such as rotating the image or scaling the image have also proven 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 ways of augmenting datasets for these tasks.
- There are also transformations that we would like our classifiers to be invariant to, but which are not easy to perform. For example, out-of-plane rotation can not be implemented as a simple geometric operation on the input pixels.
- 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. Neural networks prove not to be very robust to noise, however.
- One way to improve the robustness of neural networks is simply to train them with random noise applied to their inputs. Input noise injection is part of some unsupervised learning algorithms such as the denoising auto encoder. 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.

- When comparing machine learning benchmark results, it is important to take the effect of dataset augmentation into account. Often, hand-designed dataset augmentation schemes can dramatically reduce the generalization error of a machine learning technique.
- 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, it is necessary to make sure that both algorithms were evaluated using the same hand-designed dataset augmentation schemes.
- Suppose that algorithm A performs poorly with no dataset augmentation and algorithm B performs well when combined with numerous synthetic transformations of the input. In such a case it is likely the synthetic transformations caused the improved performance, rather than the use of machine learning algorithm B.
- Sometimes deciding whether an experiment has been properly controlled requires subjective judgment. **For example**, machine learning algorithms that inject noise into the input are performing a form of dataset augmentation. Usually, operations that are generally applicable (such as adding Gaussian noise to the input) are considered part of the machine learning algorithm, while operations that are specific to one application domain (such as randomly cropping an image) are considered to be separate pre-processing steps.

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, it is important to remember that 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 in the service of regularizing models is by adding it to the weights. This technique has been used primarily in the context of recurrent neural networks.
- This can be interpreted as a stochastic implementation of Bayesian inference over the weights. The Bayesian treatment of learning would consider the model weights to be uncertain and representable via a probability distribution that reflects this uncertainty. Adding noise to the weights is a practical, stochastic way to reflect this uncertainty.
- Noise applied to the weights can also be interpreted as equivalent (under some assumptions) to a more traditional form of regularization, encouraging stability of the function to be learned.

- Consider the regression setting, where we wish to train a function $\hat{y}(x)$ that maps a set of features x to a scalar using the least-squares cost function between the model predictions $\hat{y}(x)$ and the true values y:

$$J = E_{p(x,y)} [(\hat{y}(x) - y)^2]$$

The training set consists of m labeled examples $\{(x(1), y(1)), \ldots, (x(m), y(m))\}$.

- We now assume that with each input presentation we also include a random perturbation $\epsilon W \sim N(\epsilon; 0, \eta I)$ of the network weights. Let us imagine that we have a standard l-layer MLP. We denote the perturbed model as $\hat{y}_{\epsilon W}(x)$. Despite the injection of noise, we are still interested in minimizing the squared error of the output of the network. The objective function thus becomes:

$$\tilde{J}_W = E_{p(x,y,\epsilon W)} [(\hat{y}_{\epsilon W}(x) - y)^2]$$
$$= E_{p(x,y,\epsilon W)} [\hat{y}^2_{\epsilon W}(x) - 2y\hat{y}_{\epsilon W}(x) + y^2]$$

For small η, the minimization of J with added weight noise (with covariance ηI) is equivalent to minimization of J with an additional regularization term:

- 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.

Semi-Supervised Learning

- In the paradigm of semi-supervised learning, both unlabeled examples from P(x) and labeled examples from P (x, y) are used to estimate P (y | x) or predict y from x.
- In the context of deep learning, semi-supervised learning usually refers to learning a representation h = f (x). The goal is to learn a representation so that examples from the same class have similar representations. Unsupervised learning can provide useful cues for how to group examples in representation space.
- Examples that cluster tightly in the input space should be mapped to similar representations. A linear classifier in the new space may achieve better generalization in many cases.
- Instead of having separate unsupervised and supervised components in the model, one can construct models in which a generative model of either P (x) or P(x, y) shares parameters with a discriminative model of P(y | x).
- One can then trade-off the supervised criterion – log P(y | x) with the unsupervised or generative one (such as – log P (x) or – log P(x, y)).

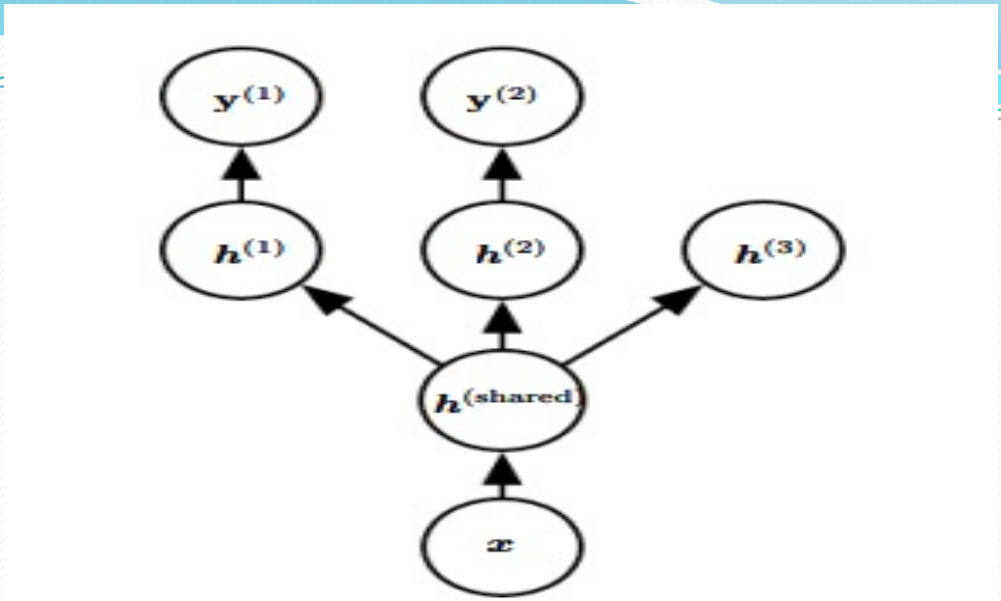
- The generative criterion then expresses a particular form of prior belief about the solution to the supervised learning problem, namely that the structure of P(x) is connected to the structure of P(y | x) in a way that is captured by the shared parameterization.
- By controlling how much of the generative criterion is included in the total criterion, one can find a better trade-off than with a purely generative or a purely discriminative training criterion.

Multi-Task Learning

- Multi-task learning is a way to improve generalization by pooling the examples arising out of several tasks.
- In the same way that additional training examples put more pressure on the parameters of the model towards values that generalize well, when part of a model is shared across tasks, that part of the model is more constrained towards good values , often yielding better generalization.
- The below is a very common form of multi-task learning, in which different supervised tasks (predicting y(i) given x) share the same input x, as well as some intermediate-level representation h(shared) capturing a common pool of factors.
- The model can generally be divided into two kinds of parts and associated parameters:

  Task-specific parameters . These are the upper layers of the neural network.

  Generic parameters, shared across all the tasks (which benefit from the pooled data of all the tasks). These are the lower layers of the neural network.



- Improved generalization and generalization error bounds can be achieved because of the shared parameters, for which statistical strength can be greatly improved (in proportion with the increased number of examples for the shared parameters, compared to the scenario of single-task models).
- Of course this will happen only if some assumptions about the statistical relationship between the different tasks are valid, meaning that there is something shared across some of the tasks.
- *Among the factors that explain the variations observed in the data associated with the different tasks, some are shared across two or more tasks.*

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 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 both to 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.

- Most hyper parameters that control model capacity have such a U-shaped validation set performance curve.
- 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. Most hyperparameters must be chosen using an expensive guess and check process, where we set a hyperparameter at the start of training, then run training for several steps to see its effect.
- The only significant cost to choosing this 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.
- If such resources are not available, then the cost of these periodic evaluations may be reduced by using a validation set that is small compared to the training set or by evaluating the validation set error less frequently and obtaining a lower resolution estimate of the optimal training time.
- 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 is a very unobtrusive form of regularization, in that it 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.
- 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 of the training data is included. There are two basic strategies one can use for this second training procedure.

Algorithm 7.2 A meta-algorithm for using early stopping to determine how long to train, then retraining on all the data.

Let $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ be the training set.
Split $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ into $(\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{X}^{(\text{valid})})$ and $(\boldsymbol{y}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{valid})})$ respectively.
Run early stopping (algorithm 7.1) starting from random $\boldsymbol{\theta}$ using $\boldsymbol{X}^{(\text{subtrain})}$ and $\boldsymbol{y}^{(\text{subtrain})}$ for training data and $\boldsymbol{X}^{(\text{valid})}$ and $\boldsymbol{y}^{(\text{valid})}$ for validation data. This returns i^*, the optimal number of steps.
Set $\boldsymbol{\theta}$ to random values again.
Train on $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ for i^* steps.

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

How early stopping acts as a regularizer

Algorithm 7.3 Meta-algorithm using early stopping to determine at what objective value we start to overfit, then continue training until that value is reached.

Let $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ be the training set.
Split $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ into $(\boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{X}^{(\text{valid})})$ and $(\boldsymbol{y}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{valid})})$ respectively.
Run early stopping (algorithm 7.1) starting from random $\boldsymbol{\theta}$ using $\boldsymbol{X}^{(\text{subtrain})}$ and $\boldsymbol{y}^{(\text{subtrain})}$ for training data and $\boldsymbol{X}^{(\text{valid})}$ and $\boldsymbol{y}^{(\text{valid})}$ for validation data. This updates $\boldsymbol{\theta}$.
$\epsilon \leftarrow J(\boldsymbol{\theta}, \boldsymbol{X}^{(\text{subtrain})}, \boldsymbol{y}^{(\text{subtrain})})$
while $J(\boldsymbol{\theta}, \boldsymbol{X}^{(\text{valid})}, \boldsymbol{y}^{(\text{valid})}) > \epsilon$ **do**
  Train on $\boldsymbol{X}^{(\text{train})}$ and $\boldsymbol{y}^{(\text{train})}$ for n steps.
end while

- Regularization by early stopping can be done either by dividing the dataset into training and test sets and then using cross-validation on the training set or by dividing the dataset into training, validation and test sets, in which case cross-validation, is not required.
- Here, the second case is analyzed. In early stopping, the algorithm is trained using the training set, and the point at which to stop training is determined from the validation set.
- Training error and validation error are analyzed. The training error steadily decreases while the validation error decreases until a point, after which it increases. This is because, during training, the learning model starts to overfit the training data.
- This causes the training error to decrease while the validation error increases. So a model with better validation set error can be obtained if the parameters that give the least validation set error are used.
- Each time the error on the validation set decreases, a copy of the model parameters is stored. When the training algorithm terminates, these parameters which give the least validation set error are finally returned and not the last modified parameters.

- In Regularization by Early Stopping, we stop training the model when the performance of the model on the validation set is getting worse-increasing loss or decreasing accuracy or poorer values of the scoring metric.
- By plotting the error on the training dataset and the validation dataset together, both the errors decrease with a number of iterations until the point where the model starts to overfit.
- After this point, the training error still decreases but the validation error increases. So, even if training is continued after this point, early stopping essentially returns the set of parameters that were used at this point and so is equivalent to stopping training at that point.
- So, the final parameters returned will enable the model to have low variance and better generalization. The model at the time the training is stopped will have a better generalization performance than the model with the least training error.
- Early stopping can be thought of as implicit regularization, contrary to regularization via weight decay. This method is also efficient since it requires less amount of training data, which is not always available.
- Due to this fact, early stopping requires lesser time for training compared to other regularization methods. Repeating the early stopping process many times may result in the model overfitting the validation dataset, just as similar as overfitting occurs in the case of training data.

- 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 that early stopping automatically determines the correct amount of regularization while weight decay requires many training experiments with different values of its hyperparameter.

Parameter Tying and Parameter Sharing

- By adding constraints or penalties to the parameters, we have always done so with respect to a fixed region or point.
- For example, L2 regularization (or weight decay) penalizes model parameters for deviating from the fixed value of zero. However, sometimes we may need other ways to express our prior knowledge about suitable values of the model parameters.
- Sometimes we might not know precisely what values the parameters should take but we know, from knowledge of the domain and model architecture, that there should be some dependencies between the model parameters.
- A common type of dependency that we often want to express is that certain parameters should be close to one another.
- Consider the following scenario: we have two models performing the same classification task (with the same set of classes) but with somewhat different input distributions.
- Formally, we have model A with parameters w(A) and model B with parameters w(B) . The two models map the input to two different, but related outputs: y^(A) = f(w(A), x) and y^(B) = g(w(B) , x).

- Let us imagine that the tasks are similar enough (perhaps with similar input and output distributions) that we believe the model parameters should be close to each other: ∀i, wi^(A)should be close to wi^(B).
- We can leverage this information through regularization. Specifically, we can use a parameter norm penalty of the form: Ω(w^(A), w^(B)) = ||w(A) – w(B)||2 2. Here we used an L2 penalty, but other choices are also possible.
- While a parameter norm penalty is one way to regularize parameters to be close to one another, the more popular way is to use constraints: to force sets of parameters to be equal.
- This method of regularization is often referred to as parameter sharing, because we interpret the various models or model components as sharing a unique set of parameters.
- A significant advantage of parameter sharing over regularizing the parameters to be close (via a norm penalty) is that only a subset of the parameters (the unique set) need to be stored in memory. In certain models—such as the convolution neural network—this can lead to significant reduction in the memory footprint of the model.

- Convolution Neural Networks By far the most popular and extensive use of parameter sharing occurs in convolution neural networks (CNNs) applied to computer vision.
- Natural images have many statistical properties that are invariant to translation. For example, a photo of a cat remains a photo of a cat if it is translated one pixel to the right.
- CNNs take this property into account by sharing parameters across multiple image locations. The same feature (a hidden unit with the same weights) is computed over different locations in the input.
- This means that we can find a cat with the same cat detector whether the cat appears at column i or column i + 1 in the image.
- Parameter sharing has allowed CNNs to dramatically lower the number of unique model parameters and to significantly increase network sizes without requiring a corresponding increase in training data. It remains one of the best examples of how to effectively incorporate domain knowledge into the network architecture.

Sparse Representations

- Weight decay acts by placing a penalty directly on the model parameters. Another strategy is to place a penalty on the activations of the units in a neural network, encouraging their activations to be sparse. This indirectly imposes a complicated penalty on the model parameters.
- Representational sparsity, on the other hand, describes a representation where many of the elements of the representation are zero (or close to zero). A simplified view of this distinction can be illustrated in the context of linear regression:

$$\begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix}$$

$\boldsymbol{y} \in \mathbb{R}^m$ $\quad$ $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ $\quad$ $\boldsymbol{x} \in \mathbb{R}^n$

- In the first expression, we have an example of a sparsely parameterized linear regression model. In the second, we have linear regression with a sparse representation h of the data x. That is, h is a function of x that, in some sense, represents the information present in x, but does so with a sparse vector.

$$\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix} = \begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix}$$

$\boldsymbol{y} \in \mathbb{R}^m$ $\quad \boldsymbol{B} \in \mathbb{R}^{m \times n}$ $\quad \boldsymbol{h} \in \mathbb{R}^n$

- Representational regularization is accomplished by the same sorts of mechanisms that we have used in parameter regularization.
- Norm penalty regularization of representations is performed by adding to the loss function J a norm penalty on the representation.
- This penalty is denoted Ω(h). As before, we denote the regularized loss function by ˜ **J:**

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

where $\alpha \in [0, \infty)$ weights the relative contribution of the norm penalty term, with larger values of α corresponding to more regularization.

- Just as an L1 penalty on the parameters induces parameter sparsity, an L1 penalty on the elements of the representation induces representational sparsity: $\boldsymbol{\Omega(h) = ||h||_1 = \sum_i |h_i|}$. Of course, the L1 penalty is only one choice of penalty that can result in a sparse representation.
- Other approaches obtain representational sparsity with a hard constraint on the activation values. For example, orthogonal matching pursuit encodes an input x with the representation h that solves the constrained optimization problem.

$$\arg\min_{\boldsymbol{h}, \|\boldsymbol{h}\|_0 < k} \|\boldsymbol{x} - \boldsymbol{W}\boldsymbol{h}\|^2,$$

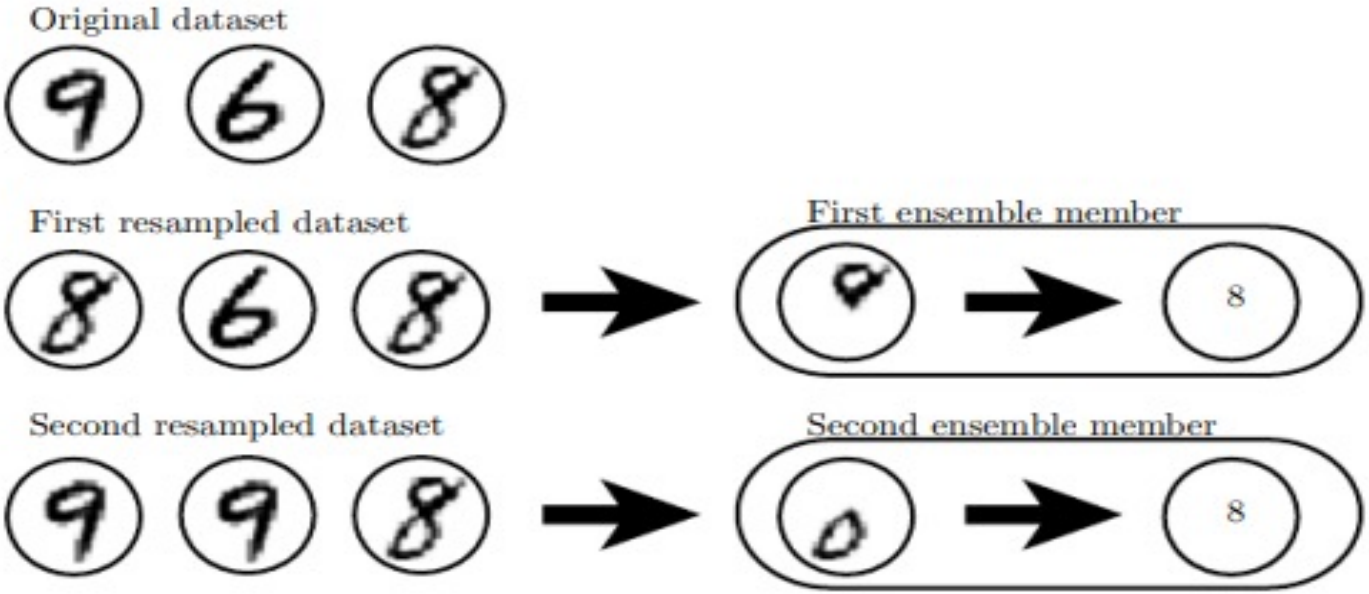
where ||h||0 is the number of non-zero entries of h. This problem can be solved efficiently when **W** is constrained to be orthogonal. This method is often called OMP-k with the value of k specified to indicate the number of non-zero features allowed.

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 of 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.
- Consider for example a set of k regression models. Suppose that each model makes an error €i on each example, with the errors drawn from a zero-mean multivariate normal distribution with variances E[€i^2] = v and covariances E[€i €j] = c. Then the error made by the average prediction of all the ensemble models is 1/k ∑i €ii. The expected squared error of the ensemble predictor is

$$\mathbb{E}\left[\left(\frac{1}{k}\sum_i \epsilon_i\right)^2\right] = \frac{1}{k^2}\mathbb{E}\left[\sum_i\left(\epsilon_i^2 + \sum_{j\neq i}\epsilon_i\epsilon_j\right)\right]$$
$$= \frac{1}{k}v + \frac{k-1}{k}c.$$

- 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 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.
- Different ensemble methods construct the ensemble of models in different ways. For example, each member of the ensemble could be formed by training a completely

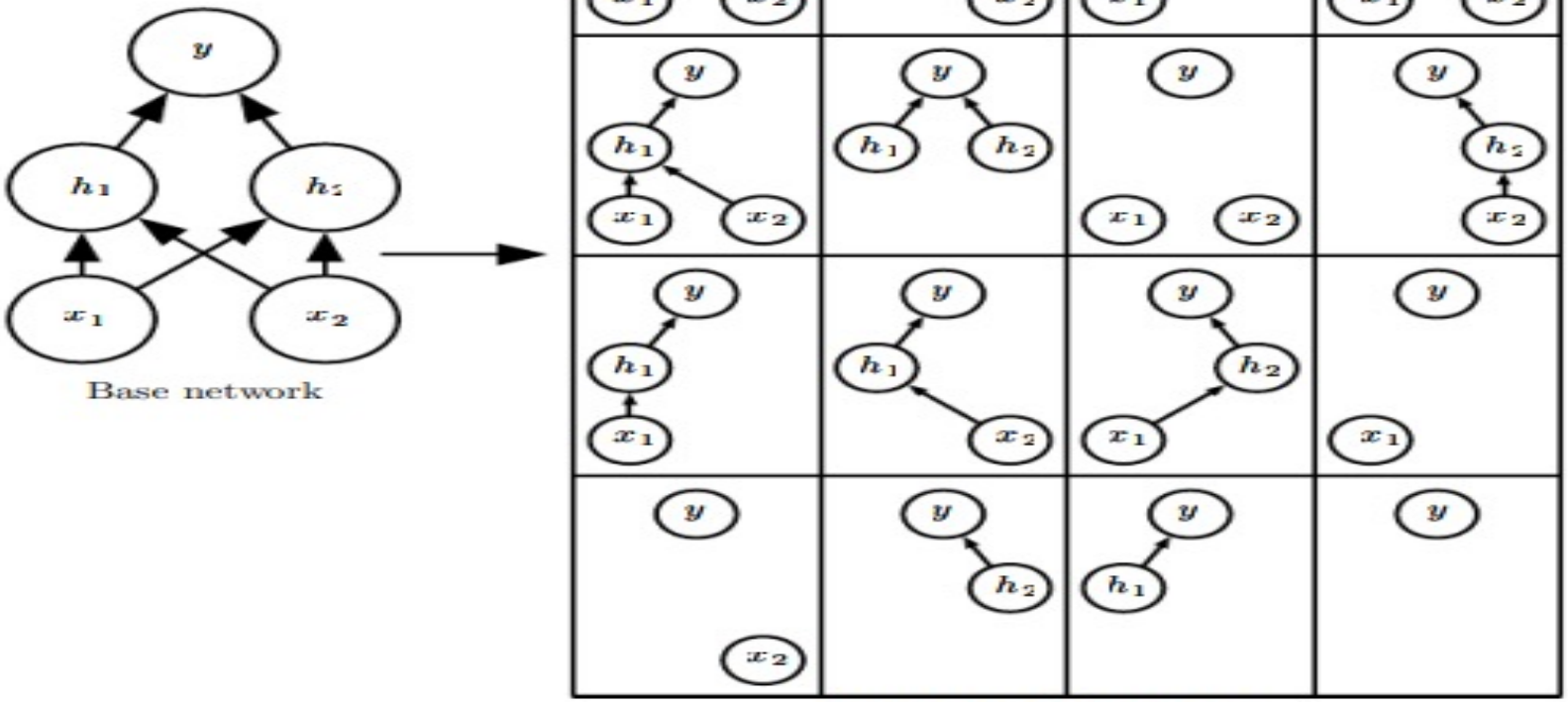


- Bagging is a method that allows the same kind of model, training algorithm and objective function to be reused several times.
- Specifically, bagging involves constructing k different datasets. Each dataset has the same number of examples as the original dataset, but each dataset is constructed by 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 also contains several duplicate examples (on average around 2/3 of the examples from the original dataset are found in the resulting training set, if it has the same size as the original).
- 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.
- 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. For this reason, benchmark comparisons are usually made using a single model.
- 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**

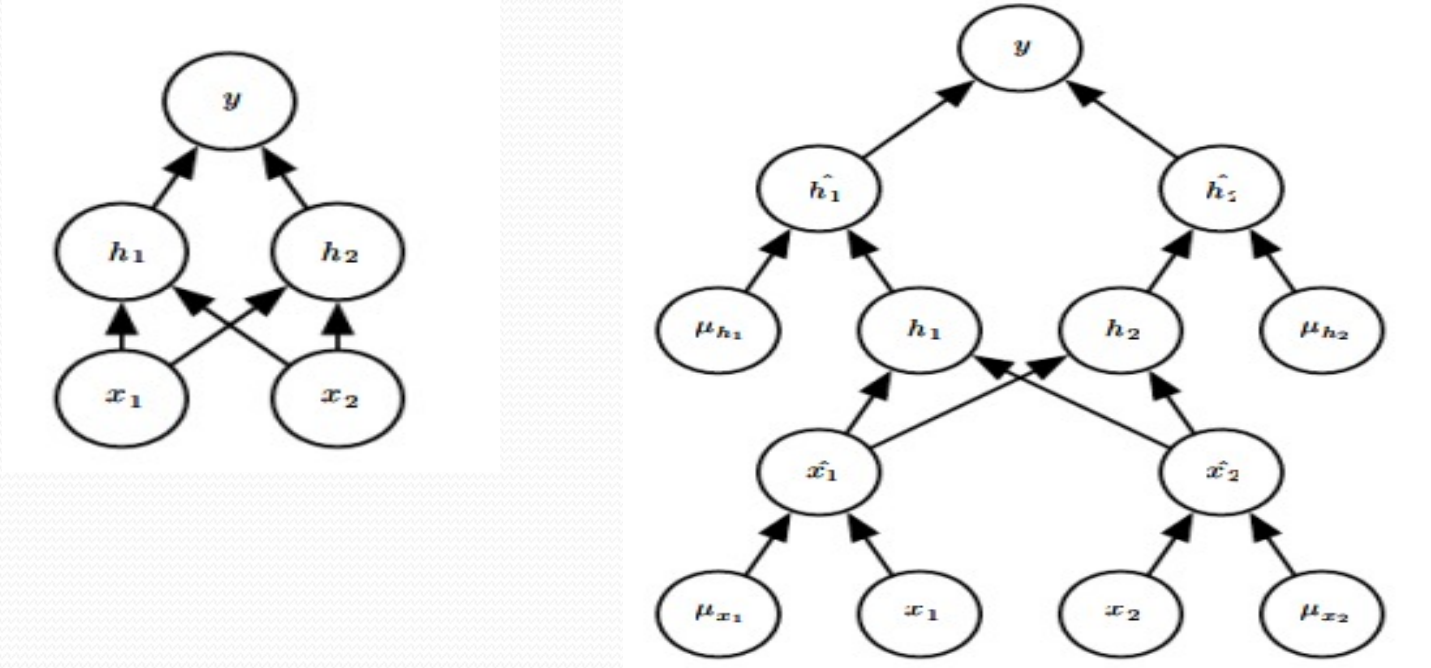
Dropout

- Dropout provides a computationally inexpensive but powerful method of regularizing a broad family of models. To a first approximation, dropout can be thought of as a method of making bagging practical for ensembles of very many large neural networks.
- Bagging involves training multiple models, and evaluating multiple models on each test example. This seems impractical when each model is a large neural network, since training and evaluating such networks is costly in terms of runtime and memory. It is common to use ensembles of five to ten neural networks.
- Dropout provides an inexpensive approximation to training and evaluating a bagged ensemble of exponentially many neural networks.
- Specifically, dropout trains the ensemble consisting of all sub-networks that can be formed by removing non-output units from an underlying base network
- In most modern neural networks, based on a series of affine transformations and nonlinearities, we can effectively remove a unit from a network by multiplying its output value by zero.

- This procedure requires some slight modification for models such as radial basis function networks, which take the difference between the unit's state and some reference value.
- Here, we present the dropout algorithm in terms of multiplication by zero for simplicity, but it can be trivially modified to work with other operations that remove a unit from the network.



- In this small example, a large proportion of the resulting networks have no input units or no path connecting the input to the output. This problem becomes insignificant for networks with wider layers, where the probability of dropping all possible paths from inputs to outputs becomes smaller.



An example of forward propagation through a feedforward network using dropout. (Top)In this example, we use a feedforward network with two input units, one hidden layer with two hidden units, and one output unit. (Bottom)To perform forward propagation with dropout, we randomly sample a vector μ with one entry for each input or hidden unit in the network. The entries of μ are binary and are sampled independently from each other. The probability of each entry being 1 is a hyperparameter, usually 0.5 for the hidden layers and 0.8 for the input. Each unit in the network is multiplied by the corresponding mask, and then forward propagation continues through the rest of the network as usual. This is equivalent to randomly selecting one of the sub-networks from figure 7.6 and running forward propagation through it.

- In the case of bagging, each model i produces a probability distribution **p^(i) (y | x)**. The prediction of the ensemble is given by the arithmetic mean of all of these distributions,

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

- In the case of dropout, each sub-model defined by mask vector μ defines a probability distribution p(y | x, μ). The arithmetic mean over all masks is given by

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

where p(μ) is the probability distribution that was used to sample μ at training time.

- Because this sum includes an exponential number of terms, it is intractable to evaluate except in cases where the structure of the model permits some form of simplification.
- However, there is an even better approach, that allows us to obtain a good approximation to the predictions of the entire ensemble, at the cost of only one forward propagation.
- The geometric mean of multiple probability distributions is not guaranteed to be a probability distribution. To guarantee that the result is a probability distribution, we impose the requirement that none of the sub-models assigns probability o to any event, and we renormalize the resulting distribution.

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

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

where d is the number of units that may be dropped. Here we use a uniform distribution over μ to simplify the presentation, but non-uniform distributions are also possible. To make predictions we must re-normalize the ensemble:

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

- key insight involved in dropout is that we can approximate p_{ensemble} by evaluating p(y | 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.
- 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.

- 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.

Adversarial Training

- In many cases, neural networks have begun to reach human performance when evaluated on an i.i.d. test set. It is natural therefore to wonder whether these models have obtained a true human-level understanding of these tasks.
- In order to probe the level of understanding a network has of the underlying task, we can search for examples that the model misclassifies.
- Neural networks that perform at human level accuracy have a nearly 100% error rate on examples that are intentionally constructed by using an optimization procedure to search for an input x^1 near a data point x such that the model output is very different at x^1 . In many cases, x^1 can be so similar to x that a human observer cannot tell the difference between the original example and the adversarial example, but the network can make highly different predictions.
- One of the primary causes of these adversarial examples is excessive linearity. Neural networks are built out of primarily linear building blocks. In some experiments the overall function they implement proves to be highly linear as a result. These linear functions are easy to optimize.

- Unfortunately, the value of a linear function can change very rapidly if it has numerous inputs. If we change each input by €, then a linear function with weights w can change by as much as €||w||1, which can be a very large amount if w is high-dimensional.
- Adversarial training discourages this highly sensitive locally linear behavior by encouraging the network to be locally constant in the neighborhood of the training data.
- Adversarial training helps to illustrate the power of using a large function family in combination with aggressive regularization. Purely linear models, like logistic regression, are not able to resist adversarial examples because they are forced to be linear.
- Neural networks are able to represent functions that can range from nearly linear to nearly locally constant and thus have the flexibility to capture linear trends in the training data while still learning to resist local perturbation.

- Adversarial examples also provide a means of accomplishing semi-supervised learning. At a point x that is not associated with a label in the dataset, the model itself assigns some label y^.
- The model's label y^ may not be the true label, but if the model is high quality, then y^ has a high probability of providing the true label. We can seek an adversarial example x^1 that causes the classifier to output a label y^1 with y^1 != y^. Adversarial examples generated using not the true label but a label provided by a trained model are called virtual adversarial examples.

Tangent Distance, Tangent Prop, and Manifold Tangent Classifier

- Many machine learning algorithms aim to overcome the curse of dimensionality by assuming that the data lies near a low-dimensional manifold.
- One of the early attempts to take advantage of the manifold hypothesis is the tangent distance algorithm. It is a non-parametric nearest-neighbor algorithm in which the metric used is not the generic Euclidean distance but one that is derived from knowledge of the manifolds near which probability concentrates.
- It is assumed that we are trying to classify examples and that examples on the same manifold share the same category. Since the classifier should be invariant to the local factors of variation that correspond to movement on the manifold, it would make sense to use as nearest-neighbor distance between points x1 and x2 the distance between the manifolds M1 and M2 to which they respectively belong.
- Although that may be computationally difficult (it would require solving an optimization problem, to find the nearest pair of points on M1 and M2), a cheap alternative that makes sense locally is to approximate Mi by its tangent plane at xi and measure the distance between the two tangents, or between a tangent plane and a point.
- That can be achieved by solving a low-dimensional linear system (in the dimension of the manifolds). Of course, this algorithm requires one to specify the tangent vectors.

- The tangent prop algorithm trains a neural net classifier with an extra penalty to make each output f(x) of the neural net locally invariant to known factors of variation.
- These factors of variation correspond to movement along the manifold near which examples of the same class concentrate. Local invariance is achieved by requiring ∇xf (x) to be orthogonal to the known manifold tangent vectors v(i) at x, or equivalently that the directional derivative of f at x in the directions v(i) be small by adding a regularization penalty Ω:

$$\Omega(f) = \sum_i \left((\nabla_{\boldsymbol{x}} f(\boldsymbol{x}))^\top \boldsymbol{v}^{(i)} \right)^2.$$

- This regularizer can of course be scaled by an appropriate hyperparameter, and, for most neural networks, we would need to sum over many outputs rather than the lone output f(x) described here for simplicity.
- As with the tangent distance algorithm, the tangent vectors are derived a priori, usually from the formal knowledge of the effect of transformations such as translation, rotation, and scaling in images. Tangent prop has been used not just for supervised learning but also in the context of reinforcement learning.

- Tangent propagation is closely related to dataset augmentation. In both cases, the user of the algorithm encodes his or her prior knowledge of the task by specifying a set of transformations that should not alter the output of the network.
- The difference is that in the case of dataset augmentation, the network is explicitly trained to correctly classify distinct inputs that were created by applying more than an infinitesimal amount of these transformations.
- Tangent propagation does not require explicitly visiting a new input point. Instead, it analytically regularizes the model to resist perturbation in the directions corresponding to the specified transformation.
- While this analytical approach is intellectually elegant, it has two major drawbacks. First, it only regularizes the model to resist infinitesimal perturbation. Explicit dataset augmentation confers resistance to larger perturbations. Second, the infinitesimal approach poses difficulties for models based on rectified linear units.
- These models can only shrink their derivatives by turning units off or shrinking their weights. They are not able to shrink their derivatives by saturating at a high value with large weights, as sigmoid or tanh units can. Dataset augmentation works well with rectified linear units because different subsets of rectified units can activate for different transformed versions of each original input.

- Tangent propagation is also related to double backprop and adversarial training Double backprop regularizes the Jacobian to be small, while adversarial training finds inputs near the original inputs and trains the model to produce the same output on these as on the original inputs.
- Tangent propagation and dataset augmentation using manually specified transformations both require that the model should be invariant to certain specified directions of change in the input.
- Double backprop and adversarial training both require that the model should be invariant to all directions of change in the input so long as the change is small. Just as dataset augmentation is the non-infinitesimal version of tangent propagation, adversarial training is the non-infinitesimal version of double backprop.