Lecture 7, Part 1: Generalization

Roger Grosse

1 Introduction

When we train a machine learning model, we don't just want it to learn to model the training data. We want it to *generalize* to data it hasn't seen before. Fortunately, there's a very convenient way to measure an algorithm's generalization performance: we measure its performance on a held-out test set, consisting of examples it hasn't seen before. If an algorithm works well on the training set but fails to generalize, we say it is *overfitting*. Improving generalization (or preventing overfitting) in neural nets is still somewhat of a dark art, but this lecture will cover a few simple strategies that can often help a lot.

1.1 Learning Goals

- Know the difference between a training set, validation set, and test set.
- Be able to reason qualitatively about how training and test error depend on the size of the model, the number of training examples, and the number of training iterations.
- Understand the motivation behind, and be able to use, several strategies to improve generalization:
  - reducing the capacity
  - early stopping
  - weight decay
  - ensembles
  - input transformations
  - stochastic regularization

2 Measuring generalization

So far in this course, we've focused on training, or optimizing, neural networks. We defined a cost function, the average loss over the training set:

$$\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(y(\mathbf{x}^{(i)}), t^{(i)}). \tag{1}$$

But we don't just want the network to get the training examples right; we also want it to generalize to novel instances it hasn't seen before.

Fortunately, there's an easy way to measure a network's generalization performance. We simply partition our data into three subsets:

- A **training set**, a set of training examples the network is trained on.
- A **validation set**, which is used to tune hyperparameters such as the number of hidden units, or the learning rate.
- A **test set**, which is used to measure the generalization performance.

There are lots of variants on this basic strategy, including something called cross-validation. Typically, these alternatives are used in situations with small datasets, i.e. less than a few thousand examples. Most applications of neural nets involve datasets large enough to split into training, validation and test sets.

The losses on these subsets are called **training, validation, and test loss**, respectively. Hopefully it's clear why we need separate training and test sets: if we train on the test data, we have no idea whether the network is correctly generalizing, or whether it's simply memorizing the training examples. It's a more subtle point why we need a separate validation set.

- We can't tune hyperparameters on the training set, because we want to choose values that will generalize. For instance, suppose we're trying to choose the number of hidden units. If we choose a very large value, the network will be able to memorize the training data, but will generalize poorly. Tuning on the training data could lead us to choose such a large value.
- We also can't tune them on the test set, because that would be "cheating." We're only allowed to use the test set once, to report the final performance. If we "peek" at the test data by using it to tune hyperparameters, it will no longer give a realistic estimate of generalization performance.[1]

The most basic strategy for tuning hyperparameters is to do a **grid search**: for each hyperparameter, choose a set of candidate values. Separately train models using all possible combinations of these values, and choose whichever configuration gives the best validation error. A closely related alternative is **random search**: train a bunch of networks using random configurations of the hyperparameters, and pick whichever one has the best validation error. The advantage of random search over grid search is as follows: suppose your model has 10 hyperparameters, but only two of them are actually important. (You don't know which two.) It's infeasible to do a grid search in 10 dimensions, but random search still ought to provide reasonable coverage of the 2-dimensional space of the important hyperparameters. On the other hand, in a scientific setting, grid search has the advantage that it's easy to reproduce the exact experimental setup.

3 Reasoning about generalization

If a network performs well on the training set but generalizes badly, we say it is **overfitting**. A network might overfit if the training set contains **accidental regularities**. For instance, if the task is to classify handwritten digits, it might happen that in the training set, all images of 9's have pixel number 122 on, while all other examples have it off. The network might

[1] Actually, there's some fascinating recent work showing that it's possible to use a test set repeatedly, as long as you add small amounts of noise to the average error. This hasn't yet become a standard technique, but it may sometime in the future. See Dwork et al., 2015, "The reusable holdout: preserving validity in adaptive data analysis."

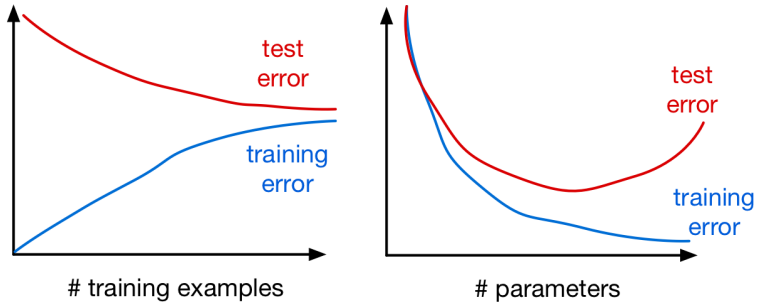

Figure 1: **(left)** Qualitative relationship between the number of training examples and training and test error. **(right)** Qualitative relationship between the number of parameters (or model capacity) and training and test error.

decide to exploit this accidental regularity, thereby correctly classifying all the training examples of 9's, without learning the true regularities. If this property doesn't hold on the test set, the network will generalize badly.

As an extreme case, remember the network we constructed in Lecture 5, which was able to learn arbitrary Boolean functions? It had a separate hidden unit for every possible input configuration. This network architecture is able to **memorize** a training set, i.e. learn the correct answer for every training example, even though it will have no idea how to classify novel instances. The problem is that this network has too large a **capacity**, i.e. ability to remember information about its training data. Capacity isn't a formal term, but corresponds roughly to the number of trainable parameters (i.e. weights). The idea is that information is stored in the network's trainable parameters, so networks with more parameters can store more information.

In order to reason qualitatively about generalization, let's think about how the training and generalization error vary as a function of the number of training examples and the number of parameters. Having more training data should only help generalization: for any particular test example, the larger the training set, the more likely there will be a closely related training example. Also, the larger the training set, the fewer the accidental regularities, so the network will be forced to pick up the true regularities. Therefore, generalization error ought to improve as we add more training examples. On the other hand, small training sets are easier to memorize than large ones, so training error tends to increase as we add more examples. As the training set gets larger, the two will eventually meet. This is shown qualitatively in Figure 1.

If the test error *increases* with the number of training examples, that's a sign that you have a bug in your code or that there's something wrong with your model.

Now let's think about the model capacity. As we add more parameters, it becomes easier to fit both the accidental and the true regularities of the training data. Therefore, training error improves as we add more parameters. The effect on generalization error is a bit more subtle. If the network has too little capacity, it generalizes badly because it fails to pick up the regularities (true or accidental) in the data. If it has too much capacity, it will memorize the training set and fail to generalize. Therefore, the effect

of capacity on test error is non-monotonic: it decreases, and then increases. We would like to design network architectures which have enough capacity to learn the true regularities in the training data, but not enough capacity to simply memorize the training set or exploit accidental regularities. This is shown qualitatively in Figure 1.

3.1 Bias and variance

For now, let's focus on squared error loss. We'd like to mathematically model the generalization error of the classifier, i.e. the expected error on examples it hasn't seen before. To formalize this, we need to introduce the **data generating distribution**, a hypothetical distribution $p_{\mathcal{D}}(\mathbf{x}, t)$ that all the training and test data are assumed to have come from. We don't need to assume anything about the form of the distribution, so the only nontrivial assumption we're making here is that the training and test data are drawn from the same distribution.

Suppose we have a test input $\mathbf{x}$, and we make a prediction y (which, for now, we treat as arbitrary). We're interested in the expected error if the targets are sampled from the conditional distribution $p_{\mathcal{D}}(t \,|\, \mathbf{x})$. By applying the properties of expectation and variance, we can decompose this expectation into two terms:

This derivation makes use of the formula $\mathrm{Var}[z] = \mathbb{E}[z^2] - \mathbb{E}[z]^2$ for a random variable z.

$$\begin{aligned}
\mathbb{E}[(y-t)^2 \,|\, \mathbf{x}] &= \mathbb{E}[y^2 - 2yt + t^2 \,|\, \mathbf{x}] \\
&= y^2 - 2y\mathbb{E}[t \,|\, \mathbf{x}] + \mathbb{E}[t^2 \,|\, \mathbf{x}] && \text{by linearity of expectation} \\
&= y^2 - 2y\mathbb{E}[t \,|\, \mathbf{x}] + \mathbb{E}[t \,|\, \mathbf{x}]^2 + \mathrm{Var}[t \,|\, \mathbf{x}] && \text{by the formula for variance} \\
&= (y - \mathbb{E}[t \,|\, \mathbf{x}])^2 + \mathrm{Var}[t \,|\, \mathbf{x}] \\
&\triangleq (y - y_\star)^2 + \mathrm{Var}[t \,|\, \mathbf{x}],
\end{aligned}$$

where in the last step we introduce $y_\star = \mathbb{E}[t \,|\, \mathbf{x}]$, which is the best possible prediction we can make, because the first term is nonnegative and the second term doesn't depend on y. The second term is known as the **Bayes error**, and corresponds to the best possible generalization error we can achieve even if we model the data perfectly.

Now let's treat y as a random variable. Assume we repeat the following experiment: sample a training set randomly from $p_{\mathcal{D}}$, train our network, and compute its predictions on $\mathbf{x}$. If we suppress the dependence on $\mathbf{x}$ for simplicity, the expected squared error decomposes as:

$$\begin{aligned}
\mathbb{E}[(y-t)^2] &= \mathbb{E}[(y - y_\star)^2] + \mathrm{Var}(t) \\
&= \mathbb{E}[y_\star^2 - 2y_\star y + y^2] + \mathrm{Var}(t) \\
&= y_\star^2 - 2y_\star\mathbb{E}[y] + \mathbb{E}[y^2] + \mathrm{Var}(t) && \text{by linearity of expectation} \\
&= y_\star^2 - 2y_\star\mathbb{E}[y] + \mathbb{E}[y]^2 + \mathrm{Var}(y) + \mathrm{Var}(t) && \text{by the formula for variance} \\
&= \underbrace{(y_\star - \mathbb{E}[y])^2}_{\text{bias}} + \underbrace{\mathrm{Var}(y)}_{\text{variance}} + \underbrace{\mathrm{Var}(t)}_{\text{Bayes error}}
\end{aligned}$$

The first term is the **bias**, which tells us how far off the model's average prediction is. The second term is the **variance**, which tells us about the variability in its predictions as a result of the choice of training set, i.e. the

amount to which it overfits the idiosyncrasies of the training data. The third term is the Bayes error, which we have no control over. So this decomposition is known as the **bias-variance decomposition**.

To visualize this, suppose we have two test examples, with targets $(t^{(1)}, t^{(2)})$. Figure 2 is a visualization in **output space**, where the axes correspond to the outputs of the network on these two examples. It shows the test error as a function of the predictions on these two test examples; because we're measuring mean squared error, the test error takes the shape of a quadratic bowl. The various quantities computed above can be seen in the diagram:

Understand why output space is different from input space or weight space.

- The generalization error is the average squared length $\|\mathbf{y} - \mathbf{t}\|^2$ of the line segment labeled *residual*.
- The bias term is the average squared length $\|\mathbb{E}[\mathbf{y}] - \mathbf{y}_*\|^2$ of the line segment labeled *bias*.
- The variance term is the spread in the green x's.
- The Bayes error is the spread in the black x's.

4 Reducing overfitting

Now that we've talked about generalization error and how to measure it, let's see how we can improve generalization by reducing overfitting. Notice that I said *reduce*, rather than *eliminate*, overfitting. Good models will probably still overfit at least a little bit, and if we try to eliminate overfitting, i.e. eliminate the gap between training and test error, we'll probably cripple our model so that it doesn't learn anything at all. Improving generalization is somewhat of a dark art, and there are very few techniques which both work well in practice and have rigorous theoretical justifications. In this section, I'll outline a few tricks that seem to help a lot. In practice, most good neural networks combine several of these tricks. Unfortunately, for the most part, these intuitive justifications are hard to translate into rigorous guarantees.

4.1 Reducing capacity

Remember the nonmonotonic relationship between model capacity and generalization error from Figure 1? This immediately suggests a strategy: there are various hyperparameters which affect the capacity of a network, such as the number of layers, or the number of units per layer. We can tune these parameters on a validation set in order to find the sweet spot, which has enough capacity to learn the true regularities, but not enough to overfit. (We can do this tuning with grid search or random search, as described above.)

A network with L layers and H units per layer will have roughly LH^2 weights. Think about why this is.

Besides reducing the number of layers or the number of units per layer, another strategy is to reduce the number of parameters by adding a **bottleneck layer**. This is a layer with fewer units than the layers below or above it. As shown in Figure 3, this can reduce the total number of connections, and hence the number of parameters.

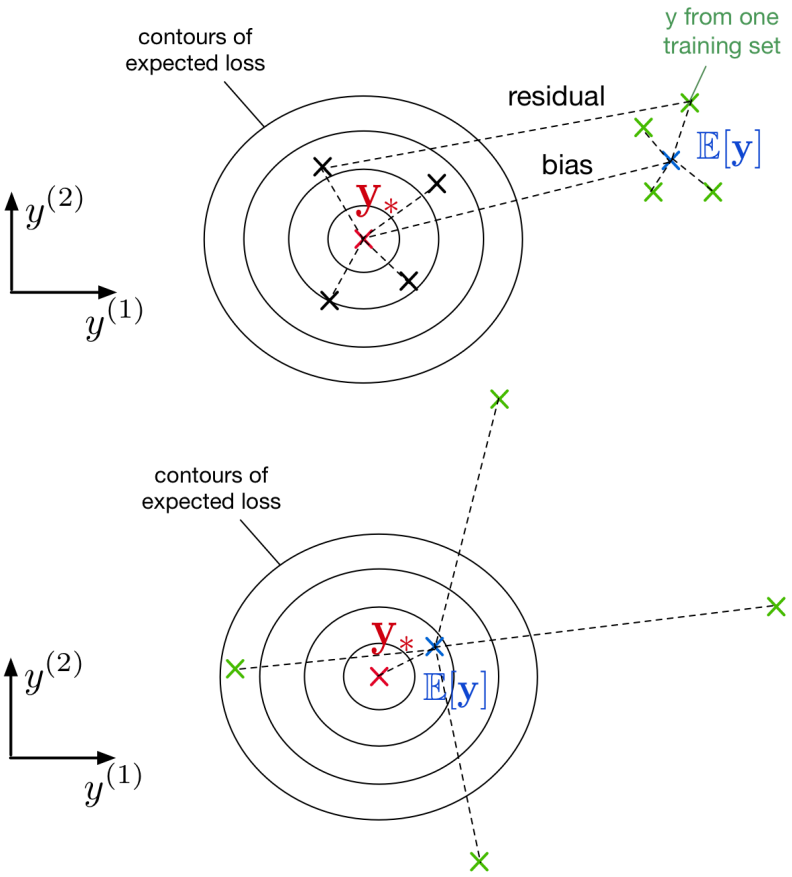

Figure 2: Schematic relating bias, variance, and error. **Top:** If the model is underfitting, the bias will be large, but the variance (spread of the green x's) will be small. **Bottom:** If the model is overfitting, the bias will be small, but the variance will be large.

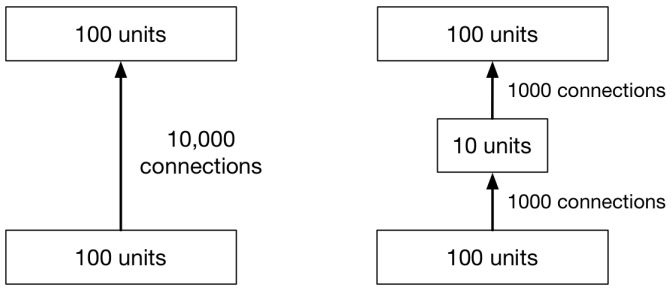

Figure 3: An example of reducing the number of parameters by inserting a linear bottleneck layer.

In general, linear and nonlinear layers have different uses. Recall that adding nonlinear layers can increase the expressive power of a network architecture, i.e. broaden the set of functions it's able to represent. By contrast, adding linear layers can't increase the expressivity, because the same function can be represented by a single layer. For instance, in Figure 3, the left-hand network can represent all the same functions as the right-hand one, since one can set $\tilde{\mathbf{W}} = \mathbf{W}^{(2)}\mathbf{W}^{(1)}$; it can also represent some functions that the right-hand one can't. The main use of linear layers, therefore, is for bottlenecks. One benefit is to reduce the number of parameters, as described above. Bottlenecks are also useful for another reason which we'll talk about later on, when we discuss autoencoders.

Reducing capacity has an important drawback: it might make the network too simple to learn the true regularities in the data. Therefore, it's often preferable to keep the capacity high, but prevent it from overfitting in other ways. We'll discuss some such alternatives now.

4.2 Early stopping

Think about how the training and test error change over the course of training. Clearly, the training error ought to continue improving, since we're optimizing the training error. (If you find the training error going up, there may be something wrong with your optimizer.) The test error generally improves at first, but it may eventually start to increase as the network starts to overfit. Such a pattern is shown in Figure 4. (Curves such as these are referred to as **training curves**.) This suggests an obvious strategy: stop the training at the point where the generalization error starts to increase. This strategy is known as **early stopping**. Of course, we can't do early stopping using the test set, because that would be cheating. Instead, we would determine when to stop by monitoring the validation error during training.

Unfortunately, implementing early stopping is a bit harder than it looks from this cartoon picture. The reason is that the training and validation error fluctuate during training (because of stochasticity in the gradients), so it can be hard to tell whether an increase is simply due to these fluctuations. One common heuristic is to space the validation error measurements far apart, e.g. once per epoch. If the validation error fails to improve after one epoch (or perhaps after several consecutive epochs), then we stop training. This heuristic isn't perfect, and if we're not careful, we might stop training too early.

4.3 Regularization and weight decay

So far, all of the cost functions we've discussed have consisted of the average of some loss function over the training set. Often, we want to add another term, called a **regularization term**, or **regularizer**, which penalizes hypotheses we think are somehow pathological and unlikely to generalize well.

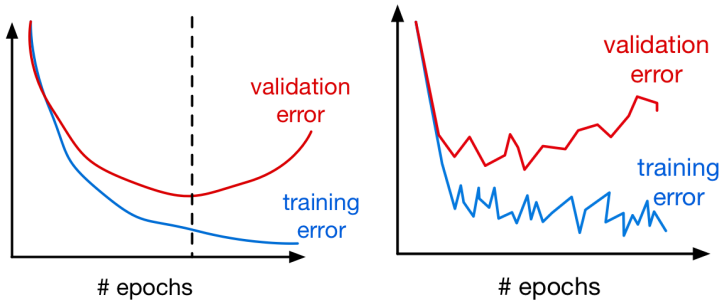

Figure 4: Training curves, showing the relationship between the number of training iterations and the training and test error. **(left)** Idealized version. **(right)** Accounting for fluctuations in the error, caused by stochasticity in the SGD updates.

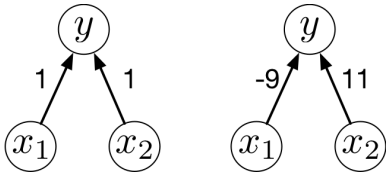

Figure 5: Two sets of weights which make the same predictions assuming inputs x_1 and x_2 are identical.

The total cost, then, is

$$\mathcal{J}(\boldsymbol{\theta}) = \underbrace{\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(y(\mathbf{x},\boldsymbol{\theta}),t)}_{\text{training loss}} + \underbrace{\mathcal{R}(\boldsymbol{\theta})}_{\text{regularizer}} \tag{2}$$

For instance, suppose we are training a linear regression model with two inputs, x_1 and x_2, and these inputs are identical in the training set. The two sets of weights shown in Figure 5 will make identical predictions on the training set, so they are equivalent from the standpoint of minimizing the loss. However, Hypothesis A is somehow better, because we would expect it to be more stable if the data distribution changes. E.g., suppose we observe the input $(x_1 = 1, x_2 = 0)$ on the test set; in this case, Hypothesis A will predict 1, while Hypothesis B will predict -8. The former is probably more sensible. We would like a regularizer to favor Hypothesis A by assigning it a smaller penalty.

One such regularizer which achieves this is L_2 **regularization**; for a linear model, it is defined as follows:

This is an abuse of terminology; mathematically speaking, this really corresponds to the *squared* L_2 norm.

$$\mathcal{R}_{L_2}(\mathbf{w}) = \frac{\lambda}{2}\sum_{j=1}^{D} w_j^2. \tag{3}$$

(The hyperparameter λ is sometimes called the **weight cost**.) L_2 regularization tends to favor hypotheses where the norms of the weights are

smaller. For instance, in the above example, with $\lambda = 1$, it assigns a penalty of $\frac{1}{2}(1^2+1^2) = 1$ to Hypothesis A and $\frac{1}{2}((-8)^2+10^2) = 82$ to Hypothesis B, so it strongly prefers Hypothesis A. Because the cost function includes both the training loss and the regularizer, the training algorithm is encouraged to find a compromise between the fit to the training data and the norms of the weights. L_2 regularization can be generalized to neural nets in the obvious way: penalize the sum of squares of all the weights in all layers of the network.

It's pretty straightforward to incorporate regularizers into the stochastic gradient descent computations. In particular, by linearity of derivatives,

$$\frac{\partial \mathcal{J}}{\partial \theta_j} = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j} + \frac{\partial \mathcal{R}}{\partial \theta_j}. \tag{4}$$

If we derive the SGD update in the case of L_2 regularization, we get an interesting interpretation.

Observe that in SGD, the regularizer derivatives do not need to be estimated stochastically.

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial \mathcal{J}^{(i)}}{\partial \theta_j} \tag{5}$$

$$= \theta_j - \alpha \left(\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j} + \frac{\partial \mathcal{R}}{\partial \theta_j} \right) \tag{6}$$

$$= \theta_j - \alpha \left(\frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j} + \lambda \theta_j \right) \tag{7}$$

$$= (1 - \alpha\lambda)\theta_j - \alpha \frac{\partial \mathcal{L}^{(i)}}{\partial \theta_j}. \tag{8}$$

In each iteration, we shrink the weights by a factor of $1 - \alpha\lambda$. For this reason, L_2 regularization is also known as **weight decay**.

Regularization is one of the most fundamental concepts in machine learning, and tons of theoretical justifications have been proposed. Regularizers are sometimes viewed as penalizing the "complexity" of a network, or favoring explanations which are "more likely." One can formalize these viewpoints in some idealized settings. However, these explanations are very difficult to make precise in the setting of neural nets, and they don't explain a lot of the phenomena we observe in practice. For these reasons, I won't attempt to justify weight decay beyond the explanation I just provided.

4.4 Ensembles

Think back to Figure 2. If you average the predictions of multiple networks trained independently on separate training sets, this reduces the variance of the predictions, which can lead to lower loss. Of course, we can't actually carry out the hypothetical procedure of sampling training sets independently (otherwise we're probably better off combining them into one big training set). We could try to train a bunch of networks on the *same* training set starting from different initializations, but their predictions might be too similar to get much benefit from averaging. However, we can try to simulate the effect of independent training sets by somehow injecting variability into the training procedure. Here some ways of injecting variability:

- Train on random subsets of the full training data. This procedure is known as **bagging**.
- Train networks with different architectures (e.g. different numbers of layers or units, or different choice of activation function).
- Use entirely different models or learning algorithms.

The set of trained models whose predictions we're combining is known as an **ensemble**. Ensembles of networks often generalize quite a bit better than single networks. This benefit is significant enough that the winning entries for most of the major machine learning competitions (e.g. ImageNet, Netflix, etc.) used ensembles.

It's possible to prove that ensembles outperform individual networks in the case of convex loss functions. In particular, suppose the loss function $\mathcal{L}$ is convex as a function of the outputs $\mathbf{y}$. Then, by the definition of convexity,

This isn't the same as the cost being convex as a function of $\boldsymbol{\theta}$, which we saw can't happen for MLPs. Lots of loss functions are convex with respect to $\mathbf{y}$, such as squared error or cross-entropy.

$$\mathcal{L}(\lambda_1 y_1 + \cdots + \lambda_N y_N, t) \leq \lambda_1 \mathcal{L}(y_1, t) + \cdots + \lambda_N \mathcal{L}(y_N, t) \quad \text{for } \lambda_i \geq 0, \sum_i \lambda_i = 1. \tag{9}$$

Hence, the average of the predictions must beat the average losses of the individual predictions. Note that this is true regardless of where the $\mathbf{y}$s came from. They could be outputs of different neural networks, or completely different learning algorithms, or even numbers you pulled out of a hat. The guarantee doesn't hold for non-convex cost functions (such as error rate), but ensembles still tend to be very effective in practice.

This result is closely related to the Rao-Blackwell theorem from statistics.

4.5 Data augmentation

Another trick is to artificially augment the training set by introducing distortions into the inputs, a procedure known as **data augmentation**. This is most commonly used in vision applications. Suppose we're trying to classify images of objects, or of handwritten digits. Each time we visit a training example, we can randomly distort it, for instance by shifting it by a few pixels, adding noise, rotating it slightly, or applying some sort of warping. This can increase the effective size of the training set, and make it more likely that any given test example has a closely related training example. Note that the class of useful transformations will depend on the task; for instance, in object recognition, it might be advantageous to flip images horizontally, whereas this wouldn't make sense in the case of handwritten digit classification.

4.6 Stochastic regularization

One of the biggest advances in neural networks in the past few years is the use of stochasticity to improve generalization. So far, all of the network architectures we've looked at compute functions deterministically. But by injecting some stochasticity into the computations, we can sometimes prevent certain pathological behaviors and make it hard for the network to overfit. We tend to call this **stochastic regularization**, even though it doesn't correspond to adding a regularization term to the cost function.

The most popular form of stochastic regularization is **dropout**. The algorithm itself is simple: we drop out each individual unit with some probability ρ (usually $\rho = 1/2$) by setting its activation to zero. We can represent this in terms of multiplying the activations by a **mask variable** m_i, which randomly takes the values 0 or 1:

$$h_i = m_i \cdot \phi(z^{(i)}). \tag{10}$$

We derive the backprop equations in the usual way:

$$\overline{z^{(i)}} = \overline{h_i} \cdot \frac{\mathrm{d}h_i}{\mathrm{d}z^{(i)}} \tag{11}$$

$$= \overline{h_i} \cdot m_i \cdot \phi'(z^{(i)}) \tag{12}$$

Why does dropout help? Think back to Figure 5, where we had two different sets of weights which make the same predictions if inputs x_1 and x_2 are always identical. We saw that L_2 regularization strongly prefers A over B. Dropout has the same preference. Suppose we drop out each of the inputs with 1/2 probability. B's predictions will vary wildly, causing it to get much higher error on the training set. Thus, it can achieve some of the same benefits that L_2 regularization is intended to achieve.

One important point: while stochasticity is helpful in preventing overfitting, we don't want to make predictions stochastically at test time. One naïve approach would be to simply not use dropout at test time. Unfortunately, this would mean that all the units receive twice as many incoming signals as they do during training time, so their responses will be very different. Therefore, at test time, we compensate for this by multiplying the values of the weights by $1 - \rho$. You'll see an interesting interpretation of this in Homework 4.

In a few short years, dropout has become part of the standard toolbox for neural net training, and can give a significant performance boost, even if one is already using the other techniques described above. Other stochastic regularizers have also been proposed; notably batch normalization, a method we already mentioned in the context of optimization, but which has also been shown to have some regularization benefits. It's also been observed that the stochasticity in stochastic gradient descent (which is normally considered a drawback) can itself serve as a regularizer. The details of stochastic regularization are still poorly understood, but it seems likely that it will continue to be a useful technique.