**Chapter 11**

**PRACTICAL METHODOLOGY**

* Successfully applying deep learning techniques requires more than just a good knowledge of what algorithms exist.
* During day to day development of machine learning systems, practitioners need to decide whether to gather more data, increase or decrease model capacity, add or remove regularizing features, improve the optimization of a model, improve approximate inference in a model, or debug the software implementation of the model.
* We recommend the following practical design process:

1. Determine your goals—what error metric to use, and your target value for this error metric.
2. Establish a working end-to-end pipeline as soon as possible, including the estimation of the appropriate performance metrics.
3. Diagnose which components are performing worse than expected and whether it is due to overfitting, underfitting, or a defect in the data or software.
4. Repeatedly make incremental changes such as gathering new data, adjusting hyperparameters, or changing algorithms, based on specific findings from your instrumentation.
   1. **Performance Metrics**

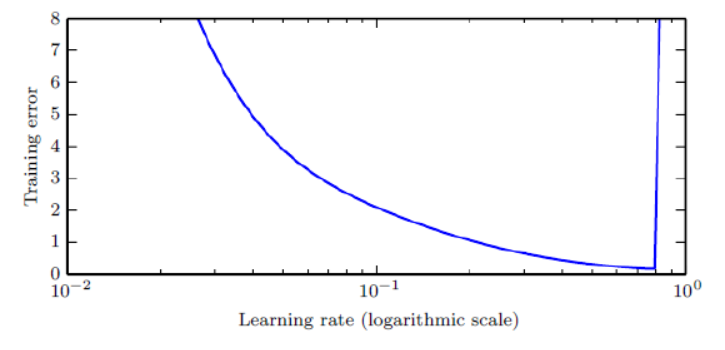
* Keep in mind that for most applications, it is impossible to achieve absolute zero error. The Bayes error defines the minimum error rate that you can hope to achieve, even if you have infinite training data and can recover the true probability distribution. This is because your input features may not contain complete information about the output variable, or because the system might be intrinsically stochastic. You will also be limited by having a finite amount of training data.
* How can one determine a reasonable level of performance to expect? Typically, in the academic setting, we have some estimate of the error rate that is attainable based on previously published benchmark results.
* Another important consideration besides the target value of the performance metric is the choice of which metric to use.
* Sometimes it is much more costly to make one kind of a mistake than another. For example, an e-mail spam detection system can make two kinds of mistakes: incorrectly classifying a legitimate message as spam, and incorrectly allowing a spam message to appear in the inbox. It is much worse to block a legitimate message than to allow a questionable message to pass through. Rather than measuring the error rate of a spam classifier, we may wish to measure some form of total cost, where the cost of blocking legitimate messages is higher than the cost of allowing spam messages.
* Sometimes we wish to train a binary classifier that is intended to detect some rare event. For example, we might design a medical test for a rare disease. Suppose that only one in every million people has this disease. We can easily achieve 99.9999% accuracy on the detection task, by simply hard-coding the classifier to always report that the disease is absent. Clearly, accuracy is a poor way to characterize the performance of such a system. One way to solve this problem is to instead measure precision and recall. Precision is the fraction of detections reported by the model that were correct, while recall is the fraction of true events that were detected. A detector that says no one has the disease would achieve perfect precision, but zero recall.
* When using precision and recall, it is common to plot a PR curve, with precision on the y-axis and recall on the x-axis. The classifier generates a score that is higher if the event to be detected occurred.
* In many cases, we wish to summarize the performance of the classifier with a single number rather than a curve. To do so, we can convert precision p and recall r into an F-score given by
* In some applications, it is possible for the machine learning system to refuse to make a decision. This is useful when the machine learning algorithm can estimate how confident it should be about a decision, especially if a wrong decision can be harmful and if a human operator is able to occasionally take over.
* Of course, the machine learning system is only useful if it is able to dramatically reduce the amount of photos that the human operators must process. A natural performance metric to use in this situation is coverage. Coverage is the fraction of examples for which the machine learning system is able to produce a response. It is possible to trade coverage for accuracy. One can always obtain 100% accuracy by refusing to process any example, but this reduces the coverage to 0%.
* Many other metrics are possible. We can for example, measure click-through rates, collect user satisfaction surveys, and so on.
  1. **Default Baseline Models**
* After choosing performance metrics and goals, the next step in any practical application is to establish a reasonable end-to-end system as soon as possible.
* If your problem has a chance of being solved by just choosing a few linear weights correctly, you may want to begin with a simple statistical model like logistic regression.
* If you know that your problem falls into an “AI-complete” category like object recognition, speech recognition, machine translation, and so on, then you are likely to do well by beginning with an appropriate deep learning model.
* First, choose the general category of model based on the structure of your data. If you want to perform supervised learning with fixed-size vectors as input, use a feedforward network with fully connected layers. If the input has known topological structure (for example, if the input is an image), use a convolutional network. In these cases, you should begin by using some kind of piecewise linear unit (ReLUs or their generalizations like Leaky ReLUs, PreLus and maxout). If your input or output is a sequence, use a gated recurrent net (LSTM or GRU).
* A reasonable choice of optimization algorithm is SGD with momentum with a decaying learning rate (popular decay schemes that perform better or worse on different problems include decaying linearly until reaching a fixed minimum learning rate, decaying exponentially, or decreasing the learning rate by a factor of 2-10 each time validation error plateaus). Another very reasonable alternative is Adam.
* Batch normalization can have a dramatic effect on optimization performance, especially for convolutional networks and networks with sigmoidal nonlinearities. While it is reasonable to omit batch normalization from the very first baseline, it should be introduced quickly if optimization appears to be problematic.
* Unless your training set contains tens of millions of examples or more, you should include some mild forms of regularization from the start. Early stopping should be used almost universally. Dropout is an excellent regularizer that is easy to implement and compatible with many models and training algorithms. Batch normalization also sometimes reduces generalization error and allows dropout to be omitted, due to the noise in the estimate of the statistics used to normalize each variable.
* A common question is whether to begin by using unsupervised learning. Some domains, such as natural language processing, are known to benefit tremendously from unsupervised learning techniques such as learning unsupervised word embeddings. In other domains, such as computer vision, current unsupervised learning techniques do not bring a benefit, except in the semi-supervised setting, when the number of labeled examples is very small (Kingma *et al.*, 2014; Rasmus *et al.*, 2015). If your application is in a context where unsupervised learning is known to be important, then include it in your first end-to-end baseline. Otherwise, only use unsupervised learning in your first attempt if the task you want to solve is unsupervised. You can always try adding unsupervised learning later if you observe that your initial baseline overfits.
  1. **Determining Whether to Gather More Data**
* After the first end-to-end system is established, it is time to measure the performance of the algorithm and determine how to improve it. Many machine learning novices are tempted to make improvements by trying out many different algorithms. However, it is often much better to gather more data than to improve the learning algorithm.
* First, determine whether the performance on the training set is acceptable. If performance on the training set is poor, the learning algorithm is not using the training data that is already available, so there is no reason to gather more data. Instead, try increasing the size of the model by adding more layers or adding more hidden units to each layer. Also, try improving the learning algorithm, for example by tuning the learning rate hyperparameter.
* If large models and carefully tuned optimization algorithms do not work well, then the problem might be the qualityof the training data. The data may be too noisy or may not include the right inputs needed to predict the desired outputs.
* If the performance on the training set is acceptable, then measure the performance on a test set. If the performance on the test set is also acceptable, then there is nothing left to be done. If test set performance is much worse than training set performance, then gathering more data is one of the most effective solutions.
* When deciding whether to gather more data, it is also necessary to decide how much to gather. It is helpful to plot curves showing the relationship between training set size and generalization error.
  1. **Selecting Hyperparameters**
* There are two basic approaches to choosing these hyperparameters:

1. Choosing them manually.
2. Choosing them automatically.

* Choosing the hyperparameters manually requires understanding what the hyperparameters do and how machine learning models achieve good generalization. Automatic hyperparameter selection algorithms greatly reduce the need to understand these ideas, but they are often much more computationally costly.

**1.4.1) Manual Hyperparameter Tuning**

* The goal of manual hyperparameter search is usually to find the lowest generalization error subject to some runtime and memory budget.
* The primary goal of manual hyperparameter search is to adjust the effective capacity of the model to match the complexity of the task. Effective capacity is constrained by three factors: the representational capacity of the model, the ability of the learning algorithm to successfully minimize the cost function used to train the model, and the degree to which the cost function and training procedure regularize the model. A model with more layers and more hidden units per layer has higher representational capacity—it is capable of representing more complicated functions.
* The generalization error typically follows a U-shaped curve when plotted as a function of one of the hyperparameters, At one extreme, the hyperparameter value corresponds to low capacity, and generalization error is high because training error is high. This is the underfitting regime. At the other extreme, the hyperparameter value corresponds to high capacity, and the generalization error is high because the gap between training and test error is high. Somewhere in the middle lies the optimal model capacity, which achieves the lowest possible generalization error.
* For some hyperparameters, overfitting occurs when the value of the hyperparameter is large. The number of hidden units in a layer is one such example, because increasing the number of hidden units increases the capacity of the model. For some hyperparameters, overfitting occurs when the value of the hyperparameter is small. For example, the smallest allowable weight decay coefficient of zero corresponds to the greatest effective capacity of the learning algorithm.
* Not every hyperparameter will be able to explore the entire U-shaped curve. Many hyperparameters are discrete, such as the number of units in a layer or the number of linear pieces in a maxout unit.
* Some hyperparameters are binary, such as a preprocessing step that normalizes the input features by subtracting their mean and dividing by their standard deviation. These hyperparameters can only explore two points on the curve. Other hyperparameters have some minimum or maximum value that prevents them from exploring some part of the curve. For example, the minimum weight decay coefficient is zero. This means that if the model is underfitting when weight decay is zero, we can not enter the overfitting region by modifying the weight decay coefficient. In other words, some hyperparameters can only subtract capacity.
* The learning rate is perhaps the most important hyperparameter. If you have time to tune only one hyperparameter, tune the learning rate. It controls the effective capacity of the model in a more complicated way than other hyperparameters—the effective capacity of the model is highest when the learning rate is correctfor the optimization problem.



***Figure:*** *Typical relationship between the learning rate and the training error.*

* If your error on the training set is higher than your target error rate, you have no choice but to increase capacity. If you are not using regularization and you are confident that your optimization algorithm is performing correctly, then you must add more layers to your network or add more hidden units. Unfortunately, this increases the computational costs associated with the model.

|  |  |  |  |
| --- | --- | --- | --- |
| **Hyperparameter** | **Increases capacity when. . .** | **Reason** | **Caveats** |
| Number of hidden  units | increased | Increasing the number of hidden units increases the representational capacity of the model. | Increasing the number of hidden units increases both the time and memory cost of essentially every operation on the model. |
| Learning rate | tuned optimally | An improper learning rate, whether too high or too low, results in a model with low effective capacity due to optimization failure |  |
| Convolution kernel  width | increased | Increasing the kernel width increases the number of parameters in the model | A wider kernel results in a narrower output dimension, reducing model capacity unless you use implicit zero padding to reduce this effect. Wider kernels require more memory for parameter storage and increase runtime, but a narrower output reduces memory cost. |
| Implicit zero padding | increased | Adding implicit zeros before convolution keeps the representation size large | Increased time and memory cost of most operations. |
| Weight decay coefficient | decreased | Decreasing the weight decay coefficient frees the model parameters to become larger |  |
| Dropout rate | decreased | Dropping units less often gives the units more opportunities to “conspire” with each other to fit the training set |  |

**1.4.2) Automatic Hyperparameter Optimization Algorithms**

* It is possible, in principle, to develop *hyperparameter optimization* algorithms that wrap a learning algorithm and choose its hyperparameters, thus hiding the hyperparameters of the learning algorithm from the user.
* Unfortunately, hyperparameter optimization algorithms often have their own hyperparameters, such as the range of values that should be explored for each of the learning algorithm’s hyperparameters.
* However, these secondary hyperparameters are usually easier to choose, in the sense that acceptable performance may be achieved on a wide range of tasks using the same secondary hyperparameters for all tasks.

**1.4.3) Grid Search**

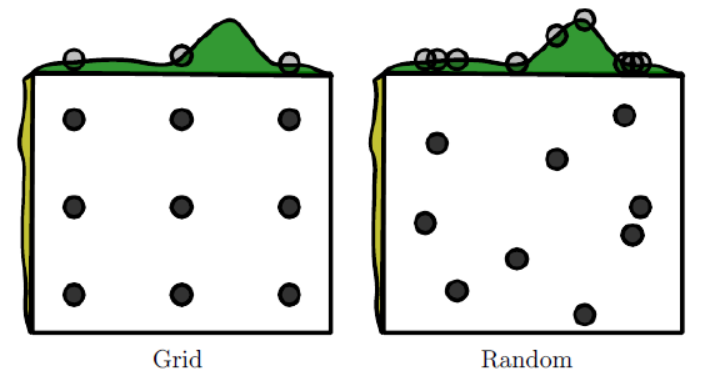
* When there are three or fewer hyperparameters, the common practice is to perform *grid search*. For each hyperparameter, the user selects a small finite set of values to explore. The grid search algorithm then trains a model for every joint specification of hyperparameter values in the Cartesian product of the set of values for each individual hyperparameter. The experiment that yields the best validation set error is then chosen as having found the best hyperparameters.
* How should the lists of values to search over be chosen? Typically, a grid search involves picking values approximately on a **logarithmic scale**, e.g., a learning rate taken within the set {.1, .01, 10−3 , 10−4 , 10−5}, or a number of hidden units taken with the set {50, 100, 200, 500, 1000, 2000}.
* Grid search usually performs best when it is performed repeatedly. For example, suppose that we ran a grid search over a hyperparameter α using values of {−1, 0, 1}. If the best value found is 1, then we underestimated the range in which the best α lies and we should shift the grid and run another search with α in, for example, {1, 2, 3}. If we find that the best value of α is 0, then we may wish to refine our estimate by zooming in and running a grid search over {−.1, 0, .1}.
* The obvious problem with grid search is that its computational cost grows exponentially with the number of hyperparameters. If there are m hyperparameters, each taking at most n values, then the number of training and evaluation trials required grows as O (nm). The trials may be run in parallel and exploit loose parallelism (with almost no need for communication between different machines carrying out the search) Unfortunately, due to the exponential cost of grid search, even parallelization may not provide a satisfactory size of search.

**1.4.4) Random Search**

* A random search proceeds as follows. First we define a marginal distribution for each hyperparameter, e.g., a Bernoulli or multinoulli for binary or discrete hyperparameters, or a uniform distribution on a log-scale for positive real-valued hyperparameters. For example,

where u(a, b ) indicates a sample of the uniform distribution in the interval (a, b). Similarly the log\_number\_of\_hidden\_units may be sampled from u(log(50), log(2000)).

* Unlike in the case of a grid search, one **should not discretize** or bin the values of the hyperparameters. This allows one to explore a larger set of values, and does not incur additional computational cost. In fact, as illustrated in below figure, a random search can be exponentially more efficient than a grid search, when there are several hyperparameters that do not strongly affect the performance measure.
* As with grid search, one may often want to run repeated versions of random search, to refine the search based on the results of the first run.
* The main reason why random search finds good solutions faster than grid search is that the there are no wasted experimental runs, unlike in the case of grid search, when two values of a hyperparameter (given values of the other hyperparameters) would give the same result. In the case of grid search, the other hyperparameters would have the same values for these two runs, whereas with random search, they would usually have different values. Hence if the change between these two values does not marginally make much difference in terms of validation set error, grid search will unnecessarily repeat two equivalent experiments while random search will still give two independent explorations of the other hyperparameters.



***Figure:*** *Comparison of grid search and random search. For illustration purposes we display two hyperparameters but we are typically interested in having many more. (Left) To perform grid search, we provide a set of values for each hyperparameter. The search algorithm runs training for every joint hyperparameter setting in the cross product of these sets. To perform random search, we provide a probability distribution (Right) over joint hyperparameter configurations. Usually most of these hyperparameters are independent from each other. Common choices for the distribution over a single hyperparameter include uniform and log-uniform (to sample from a log-uniform distribution, take the exp of a sample from a uniform distribution). The search algorithm then randomly samples joint hyperparameter configurations and runs training with each of them. Both grid search and random search evaluate the validation set error and return the best configuration.*

**1.4.5) Model-Based Hyperparameter Optimization**

* In simplified settings where it is feasible to compute the gradient of some differentiable error measure on the validation set with respect to the hyperparameters, we can simply follow this gradient (Bengio *et al.*, 1999; Bengio, 2000; Maclaurin *et al.*, 2015). Unfortunately, in most practical settings, this gradient is unavailable, either due to its high computation and memory cost, or due to hyperparameters having intrinsically non-differentiable interactions with the validation set error, as in the case of discrete-valued hyperparameters.
* To compensate for this lack of a gradient, we can build a model of the validation set error, then propose new hyperparameter guesses by performing optimization within this model. Most model-based algorithms for hyperparameter search use a Bayesian regression model to estimate both the expected value of the validation set error for each hyperparameter and the uncertainty around this expectation. Optimization thus involves a tradeoff between exploration (proposing hyperparameters for which there is high uncertainty, which may lead to a large improvement but may also perform poorly) and exploitation (proposing hyperparameters which the model is confident will perform as well as any hyperparameters it has seen so far). Contemporary approaches to hyperparameter optimization include Spearmint (Snoek *et al.*, 2012), TPE (Bergstra *et al.*, 2011) and SMAC (Hutter *et al.*, 2011).
* Bayesian hyperparameter optimization sometimes performs comparably to human experts, sometimes better, but fails catastrophically on other problems. It may be worth trying to see if it works on a particular problem but is not yet sufficiently mature or reliable.
* One drawback common to most hyperparameter optimization algorithms with more sophistication than random search is that they require for a training experiment to run to completion before they are able to extract any information from the experiment. This is much less efficient, in the sense of how much information can be gleaned early in an experiment.
* Swersky *et al.* (2014) have introduced an early version of an algorithm that maintains a set of multiple experiments. At various time points, the hyperparameter optimization algorithm can choose to begin a new experiment, to “freeze” a running experiment that is not promising, or to “thaw” and resume an experiment that was earlier frozen but now appears promising given more information.
  1. **Debugging Strategies**
* Machine learning systems are difficult to debug for a variety of reasons.
* In most cases, we do not know a priori what the intended behavior of the algorithm is. If we train a neural network on a classification task and it achieves new5% test error, we have no straightforward way of knowing if this is the expected behavior or sub-optimal behavior.
* A further difficulty is that most machine learning models have multiple parts that are each adaptive. If one part is broken, the other parts can adapt and still achieve roughly acceptable performance. For example, suppose that we are training a neural net with several layers parametrized by weights W and biases b. Suppose further that we have manually implemented the gradient descent rule for each parameter separately, and we made an error in the update for the biases:

where α is the learning rate. This erroneous update does not use the gradient at all. It causes the biases to constantly become negative throughout learning. The bug may not be apparent just from examining the output of the model though. Depending on the distribution of the input, the weights may be able to adapt to compensate for the negative biases.

* Some important debugging tests include:

1. **Visualize the model in action**

* When training a model to detect objects in images, view some images with the detections proposed by the model displayed superimposed on the image.
* When training a generative model of speech, listen to some of the speech samples it produces.

1. **Visualize the worst mistakes**

* Most models are able to output some sort of confidence measure for the task they perform. For example, classifiers based on a softmax output layer assign a probability to each class. The probability assigned to the most likely class thus gives an estimate of the confidence the model has in its classification decision.
* Typically, maximum likelihood training results in these values being overestimates rather than accurate probabilities of correct prediction, but they are somewhat useful in the sense that examples that are actually less likely to be correctly labeled receive smaller probabilities under the model.
* By viewing the training set examples that are the hardest to model correctly, one can often discover problems with the way the data has been preprocessed or labeled. For example, the Street View transcription system originally had a problem where the address number detection system would crop the image too tightly and omit some of the digits.

1. **Reasoning about software using train and test error**

* It is often difficult to determine whether the underlying software is correctly implemented. Some clues can be obtained from the train and test error.
* If training error is low but test error is high, then it is likely that that the training procedure works correctly, and the model is overfitting for fundamental algorithmic reasons.
* An alternative possibility is that the test error is measured incorrectly due to a problem with saving the model after training then reloading it for test set evaluation, or if the test data was prepared differently from the training data.
* If both train and test error are high, then it is difficult to determine whether there is a software defect or whether the model is underfitting due to fundamental algorithmic reasons. This scenario requires further tests, described next.

1. **Fit a tiny dataset**

* If you have high error on the training set, determine whether it is due to genuine underfitting or due to a software defect.
* Usually even small models can be guaranteed to be able fit a sufficiently small dataset. For example, a classification dataset with only one example can be fit just by setting the biases of the output layer correctly.
* Usually if you cannot train a classifier to correctly label a single example, an autoencoder to successfully reproduce a single example with high fidelity, or a generative model to consistently emit samples resembling a single example, there is a software defect preventing successful optimization on the training set. This test can be extended to a small dataset with few examples.

1. **Compare back-propagated derivatives to numerical derivatives**

* If you are using a software framework that requires you to implement your own gradient computations, or if you are adding a new operation to a differentiation library and must define its bprop method, then a common source of error is implementing this gradient expression incorrectly.
* One way to verify that these derivatives are correct is to compare the derivatives computed by your implementation of automatic differentiation to the derivatives computed by a finite differences. Because

we can approximate the derivative by using a small, finite :

We can improve the accuracy of the approximation by using the centered difference:

* The perturbation size must chosen to be large enough to ensure that the perturbation is not rounded down too much by finite-precision numerical computations.
* Usually, we will want to test the gradient or Jacobian of a vector-valued function

g : Rm→Rn . Unfortunately, finite differencing only allows us to take a single derivative at a time. We can either run finite differencing mn times to evaluate all of the partial derivatives of g, or we can apply the test to a new function that uses random projections at both the input and output of g. For example, we can apply our test of the implementation of the derivatives to f (x) where f (x) = uT g(vx), where u and v are randomly chosen vectors.

* Computing f ‘(x) correctly requires being able to back-propagate through g correctly, yet is efficient to do with finite differences because f has only a single input and a single output. It is usually a good idea to repeat this test for more than one value of u and v to reduce the chance that the test overlooks mistakes that are orthogonal to the random projection.
* If one has access to numerical computation on complex numbers, then there is a very efficient way to numerically estimate the gradient by using complex numbers as input to the function (Squire and Trapp, 1998). The method is based on the observation that

where i = √−1. Unlike in the real-valued case above, there is no cancellation effect due to taking the difference between the value of f at different points. This allows the use of tiny values of like = 10−150, which make the O(2) error insignificant for all practical purposes.

1. **Monitor histograms of activations and gradient**

* It is often useful to visualize statistics of neural network activations and gradients, collected over a large amount of training iterations (maybe one epoch).
* The pre-activation value of hidden units can tell us if the units saturate, or how often they do. For example, for rectifiers, how often are they off? Are there units that are always off? For tanh units, the average of the absolute value of the pre-activations tells us how saturated the unit is.
* As suggested by Bottou(2015), we would like the magnitude of parameter updates over a minibatch to represent something like 1% of the magnitude of the parameter, not 50% or 0.001% (which would make the parameters move too slowly).
* It may be that some groups of parameters are moving at a good pace while others are stalled. When the data is sparse (like in natural language), some parameters may be very rarely updated.
  1. **Example: Multi-Digit Number Recognition**
* To provide an end-to-end description of how to apply our design methodology in practice, we present a brief account of the Street View transcription system.
* The process began with data collection. The cars collected the raw data and human operators provided labels.
* The transcription project began with a choice of performance metrics and desired values for these metrics. Because maps are only useful if they have high accuracy, it was important to set a high accuracy requirement for this project. Specifically, the goal was to obtain human-level, 98% accuracy. This level of accuracy may not always be feasible to obtain. In order to reach this level of accuracy, the Street View transcription system sacrifices coverage. Coverage thus became the main performance metric optimized during the project, with accuracy held at 98%.
* After choosing quantitative goals, the next step in our recommended methodology is to rapidly establish a sensible baseline system. For vision tasks, this means a convolutional network with rectified linear units.
* In order to begin with the simplest possible baseline, the first implementation of the output layer of the model consisted of n different softmax units to predict a sequence of n characters. These softmax units were trained exactly the same as if the task were classification, with each softmax unit trained independently.
* Our recommended methodology is to iteratively refine the baseline and test whether each change makes an improvement. The first change to the Street View transcription system was motivated by a theoretical understanding of the coverage metric and the structure of the data. Specifically, the network refuses to classify an input x whenever the probability of the output sequence p(y | x) < t for some threshold t. Initially, the definition of p(y | x) was ad-hoc, based on simply multiplying all of the softmax outputs together. This motivated the development of a specialized output layer and cost function that actually computed a principled log-likelihood. This approach allowed the example rejection mechanism to function much more effectively.
* At this point, coverage was still below 90%, yet there were no obvious theoretical problems with the approach. Our methodology therefore suggests to instrument the train and test set performance in order to determine whether the problem is underfitting or overfitting. In this case, train and test set error were nearly identical.
* Because train and test set error were so similar, this suggested that the problem was either due to underfitting or due to a problem with the training data. One of the debugging strategies we recommend is to visualize the model’s worst errors. In this case, that meant visualizing the incorrect training set transcriptions that the model gave the highest confidence. These proved to mostly consist of examples where the input image had been cropped too tightly, with some of the digits of the address being removed by the cropping operation.
* This problem could have been resolved by spending weeks improving the accuracy of the address number detection system responsible for determining the cropping regions. Instead, the team took a much more practical decision, to simply expand the width of the crop region to be systematically wider than the address number detection system predicted.
* Finally, the last few percentage points of performance came from adjusting hyperparameters. This mostly consisted of making the model larger while maintaining some restrictions on its computational cost. Because train and test error remained roughly equal, it was always clear that any performance deficits were due to underfitting, as well as due to a few remaining problems with the dataset itself.