Kevin Chen

6/6/20

Improving Deep Neural Networks: Hyperparameter Tuning, Regularization, and Optimization

**Practical Aspects of Deep Learning**

Setting up your Machine Learning Application

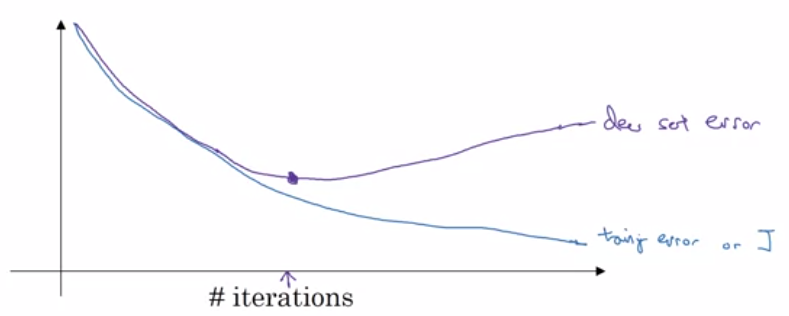
* Applied ML is a highly iterative process since it requires the selection of many hyperparameters.
  + Intuition of good hyperparameters from one application (e.g. NLP) often do not transfer well to other applications (e.g. vision, speech, recommendations/search, computer security, logistics, etc)
* Train/dev/test sets
  + Workflow: train your algorithms on the training set, select the best model on the development set (aka “dev” set, or (hold-out) cross validation set), and then evaluate your model on the test set to get an unbiased estimate of the model’s performance.
  + Previous (traditional) era of ML, splitting the data into 70/30 for train and test or 60/20/20 for train, cross-validation, and test was good. But for big data (e.g. 1 million examples), cross-validation and test sets often do not require as many examples.
  + Mismatched train/test distribution: some people train their algorithm on a different distribution than the distribution they use for dev and test sets. Example: they may train an image classifier on cat pictures from webpages (higher resolution) and use images from an app for dev/test sets (lower resolution). If this is the case, make sure dev/test sets are from the same distribution since dev set is used to optimize performance on test set.
  + Not having a test set might be okay (only dev set) if you don’t need an estimate of the model’s performance. (If this is the case, people usually refer to the dev set as their “train” set despite using it as a dev set to optimize hyperparameters.)
* Bias/variance
  + High bias = underfitting. High variance = overfitting
  + Diagnosis of bias and variance:

|  |  |  |
| --- | --- | --- |
| **Training set error** | **Dev set error:** | **Diagnosis:** |
| Low (e.g. 1%) | High (e.g. 15%) | High variance |
| High (e.g. 15%) | High (e.g. 16%) | High bias |
| High (e.g. 15%) | Even higher (e.g. 30%) | High bias and variance |

* + The above chart is under the assumption that the train and dev set distributions are the same.
  + The increase in error from train set to dev set is an indication of variance, and the amount of error in train set is indication of bias.
  + “High” error means the error is higher than the error a human would be able to achieve (also known as Bayes error). If Bayes error is 15%, then having 15% training set error and 16% would be considered good.
* Fixing high bias or high variance:
  + Start by fixing high bias. If the model has high bias, then try a bigger network, train longer (more iterations of gradient descent or an alternative optimization algorithm), another neural network architecture.
  + If you model now has high variance, get more data, try regularization, or try another neural network architecture. Now check back to see if you have high bias, and if so, repeat starting from the top.
* “Bias variance tradeoff” was applicable to the early machine learning era: lots of things you could do decreases bias at the expense of increasing variance or decreases variance at the expense of increasing bias.
  + But now, using a bigger network generally doesn’t hurt variance and using more data generally doesn’t hurt bias.

Regularizing Your Neural Network

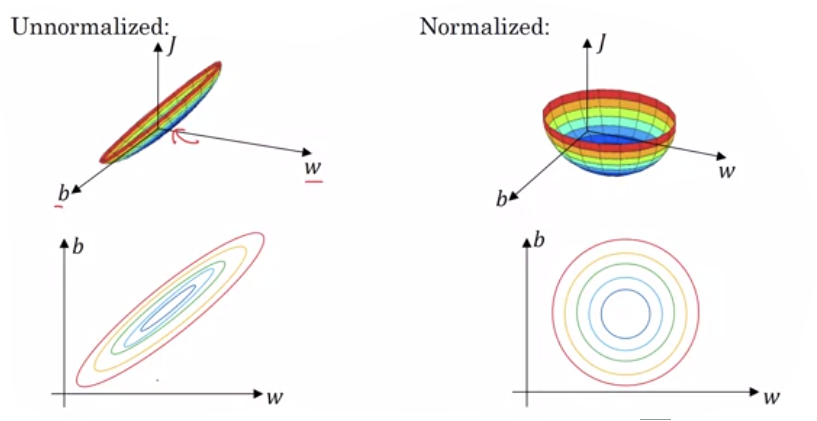
* Regularization with logistic regression:
  + “ norm” or “Euclidean norm”, which is the most common: , where is the norm of , which is equal to .
  + While you can include , most people just omit it because often has many parameters, so adding doesn’t make much difference.
  + regularization: add the term . The term is sparse (having many zeros), which makes compression easier.
  + is referred to as the regularization parameter, which is a hyperparameter tuned in the cross-validation set.
* Regularization with neural network:
  + , where is the squared norm and is equal to: . This norm is referred to as the “Frobenius norm” as opposed to the norm.
  + Gradient descent with regularization: the only difference is computing , where we add a . For example, for the vectorized version across all examples and features:
  + The above equation is known as “weight decay”, because it shrinks by a factor of , given all other things equal:
* Why does regularization prevent overfitting?
  + Regularization helps prevent overfitting by effectively reducing the impact of many of the hidden units by zeroing them out, thus resulting in a simpler network.
  + As we increase , the weights decrease. Since , this makes closer to 0. When is close to 0, it takes a smaller range of values, so the activation function of the tanh or sigmoid functions is roughly linear. If all activation functions are linear, then the entire neural network can only learn a linear function. This effectively makes the decision boundary less complicated, thus reducing overfitting.
* Dropout regularization is another technique for regularization.
  + High level idea: for each node in each layer, set a probability that we will remove that node. Then flip a coin. When removing a node, remove all incoming and outgoing edges. Then train the neural network on this smaller network. Repeat for each example.
  + We are effectively regularizing the network since the networks are smaller.
* Implementing dropout: inverted dropout technique is the most common. For each example in forward propagation, create a matrix of dimensions , where each element is a coin-flip, where the chance of flipping a “1” is the chance we’re keeping the element. Perform an element-wise multiplication between and and store that in . Divide each element in by the probability we keep the element (e.g. if we keep 50% of elements, divide by 0.5.) We do this since , so decreasing the expected value of will decrease the expected value of , but dividing by the probability we keep the elements maintains the expected value of and thus .
  + Dividing by the probability we keep the element is referred to the inverted dropout technique. This technique makes evaluating the algorithm via the test set easier since we don’t have a scaling problem.
  + For a given example, don’t zero out the same neurons between two iterations of gradient descent.
  + Don’t use drop out when making predictions during test time, as that would just add noise to the predictions.
* Why does drop-out achieve regularization?
  + Intuition 1: you work with a smaller network
  + Intuition 2: a neuron cannot rely on any one input too much, since in a given iteration, that input may be dropped. This causes it to have to spread out its weights more, which decreases or the norm of the weight matrix. As a result, it achieves regularization.
* The probability you keep a neuron in drop-out regularization can vary across layers if you worry about overfitting on some layers more than others.
  + One strategy is to make the keep probability lower on layers with more neurons since you may worry about overfitting more in these more complex layers.
  + Keep probability is usually 1.0 for input layer (most people don’t drop an input) and 1.0 for output (doesn’t make sense to not have an output).
  + Note that this approach does increase the number of hyperparameters to tune.
* Drop-out regularization has had more success in computer vision since the input feature is so large, so you often don’t have enough data and thus often overfit more.
* Drop-out regularization doesn’t have a well defined cost function , making the graph between # and iterations and less useful as a debugging tool. Hence, don’t use drop-out regularization when computing for this graph after each iteration.
* Other regularization methods:
  + Using artificial synthetic data can be used to obtain more training examples to help reduce overfitting. Example: flip an image horizontally, take random crops of the image, add distortions.
  + Early stopping: plot training error and dev set error against # of iterations. When the dev set error is minimized (i.e. before the neural network overfits with large values of ), take that optimal set of weights.



* + One downside of early stopping is that it violates the principle of orthogonalization. The problem of optimizing cost function (e.g. through gradient descent) should be tackled separately from the problem of avoiding overfitting (e.g. through regularization), but early stopping couples these two problems together. You can no longer work on these two problems separately.
  + One benefit of early stopping over regularization is that it’s simpler to use: regularization requires tuning the hyperparameter .

Setting Up Your Optimization Problem

* Normalize your inputs: subtract off the mean and divide the data by the standard deviation.
  + If you don’t scale your features, your cost function will be very elongated. This requires setting the learning rate small and thus resulting in more iterations needed for gradient descent to converge.



* One problem of training neural networks is vanishing and exploding gradients: derivatives of slopes can sometimes get very big or small.
  + The hypothesis and the gradients of the neural network may increase exponentially relative to the number of layers.
  + For example, if , then , so if ( identity matrix) and is large, then will be approximately which would be enormous. On the other hand, if , then which would mean is tiny. This argument can be extended for showing gradients will be large or tiny.
* Initializing weights to a particular distribution can partially solve the problem of vanishing and exploding gradients.
  + Since , having a large number of input features means that we want a smaller . In fact, we want if we’re using the tanh activation function, which can be done via the Python function:

np.random.rand(…dimensions…) \* np.sqrt

This is known as the Xavier initialization.

* + If you’re using a ReLU activation function, then you actually want to set the variance of to :

np.random.rand(…dimensions…) \* np.sqrt

* + Another version of the initialization uses:

np.random.rand(…dimensions…) \* np.sqrt

* + The variance you use can be a hyperparameter to tune, but is a lower priority compared to other hyperparameters.
* Gradient (grad) checking: Use a two-sided numerical approximation of gradients to verify your gradient calculations are correct.
  + The gradient of a function can be approximated by .
  + In fact, the definition of a gradient is , and it can be shown the error of the approximation is bounded by .
  + The one-sided hypothesis has an error bounded by which is much higher than for small .
* Grad checking for verifying your implementation:
  + Verify that your calculation of is approximately , where is the 3D matrix with added to and is the 3D matrix with subtracted from .
  + (The video says you need to unroll the parameters into a giant vector , and unroll into a giant vector , and then verify that . I think the above bullet point accomplishes this more succinctly.)
  + Next, you want to verify that the vector is close your numerical approximation of that vector. Do this by checking that for small , such as .
* Other notes of grad checking:
  + Don’t use in training, but instead only to debug. Checking that the gradients are approximate is computationally expensive, so just do this for the first iteration.
  + If an algorithm fails grad check, look at components to try to identify bug (e.g. is it occurring only for but not ?)
  + Remember to incorporate the regularization terms in calculating the derivatives.
  + This method doesn’t work with dropout, since it randomly drops out units each iteration. Verify implementation with dropout turned off first and then turn on dropout.
  + Run grad check again after some training. Your bug might only occur for larger values of and , so you won’t detect the error when running grad check on the first iteration where .

**Optimization Algorithms**

Mini-batch Gradient Descent

* Mini-batch gradient descent:
  + While vectorization helps speed up training, it can still take a while if is very large. You cannot make any progress until you have processed all training examples.
  + Mini-batch gradient descent lets you make progress before you have processed all the training examples.
  + We group the examples into mini-batches, where denotes the input to the th mini-batch and denotes the output of the th mini-batch. Let be the number of batches, which is equal to divided by the size of each batch, which we’ll call .
  + For example, we can group 5 million training examples into 5,000 mini-batches, each of size 1000. , etc. , etc.
* Mini-batch gradient descent algorithm:

For each iteration of gradient descent:

for :

Perform forward propagation on

// is summing across the examples in batch

Compute cost

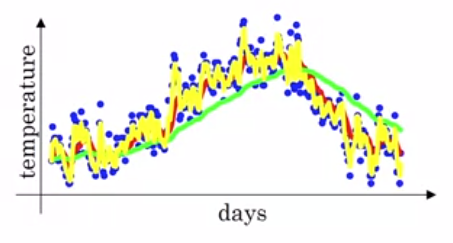
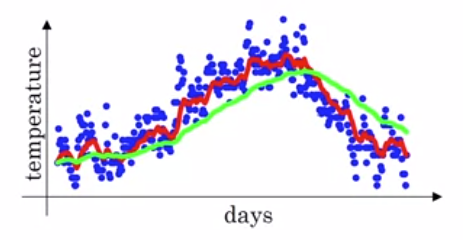
Perform backpropagation to compute gradients wrt using

,

* “1 epoch” of training is referred to as one pass through the training set. Looping over all mini-batches is 1 epoch of training.
  + One epoch of training with batch gradient descent only allows you to make one step. One epoch of training with mini-batch gradient descent lets you take steps.
  + Especially when the training set size is very large, mini-batch gradient descent runs much faster.
* The cost function should decrease on every iteration of batch gradient descent. For mini-batch gradient descent, there can be noise/oscillations, but it should generally trend downwards.
* Choosing your mini-batch size.
  + Two extremes: if mini-batch size is equal to , then you’re performing batch gradient descent. If mini-batch size is equal to 1, you’re performing stochastic gradient descent.
  + Batch gradient descent will take direct steps toward the global minimum (though slowly), whereas stochastic gradient descent will slowly meander to the global minimum.
  + Batch gradient descent will converge at local minimum (or very close to it). Stochastic gradient descent will meander around the local minimum (never fully converging).
  + In practice, you will select a mini-batch size somewhere in between the two extremes of 1 and since batch gradient descent is slow and stochastic gradient descent loses speedup from vectorization. Somewhere in between gives you the fastest learning (vectorization and make more steps per epoch).
* Reducing the learning rate over # of iterations via learning rate decay can help avoid oscillations around global minimum.
* How to choose mini-batch gradient descent:
  + Small training set : use batch gradient descent.
  + Typical mini-batch sizes are powers of 2 since code may run faster. Typical sizes are 64, 128, 256, or 512.
  + Make sure mini-batches fit in CPU/GPU memory.
  + In practice, this is another hyperparameter to tune.

Exponentially Weighted Moving Averages

* Given an ordered sequence of numerical values, , we can calculated the exponentially weighted moving average, which is defined recursively as:
  + Where is a value between 0 and 1.
* The higher is, the smoother the curve is if we plot against , and there is more latency. If is lower, then it is more noisy and susceptible to outliers. For example, red line is , green line is , and yellow line is , where is the day of year and is temperature of that day.



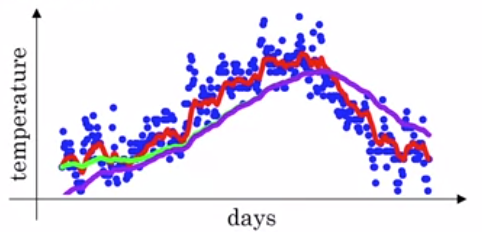
* Solving the recursion, we get that:
  + Since , this is considered to be an average across , and thus we call this a exponentially weighted moving average.
* is approximately averaging over the previous values.
  + This is derived from the fact that a value is considered insignificant if the constant in front of it is less than .
  + We note that . Hence, if we set , we get that , so that means we must raise to the in order for the value to be the smallest significant value, so thus it roughly averages across the previous values.
* Algorithm for implementing exponentially weighted moving average:

Repeat {

Get next

}

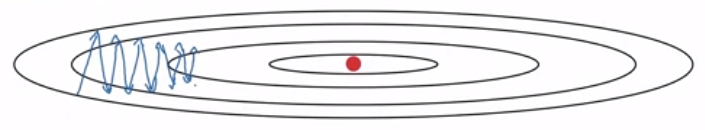
* While keeping track of the average of the previous days is more accurate, this method is computationally faster and uses less memory, which is necessary when dealing with many variables.
* We need bias correction to fix the curve for the initial values of .
  + For example, if , then since . This clearly is not a good estimate of what truly is.
  + You get the purple line instead of the green line for if we don’t use bias correction:



* + To fix this, use the value instead of . (Or alternatively, set .) We note that is the sum of the coefficients of .
* For most implementations of exponential weighted average, people often don’t bother to implement bias correction, but this is useful if you are concerned about the initial phase.

Gradient Descent with Momentum

* Gradient descent with Momentum is sometimes shortened to Momentum.
* Gradient Descent with Momentum high level idea: compute an exponentially weighted average of your gradients, and use that average to update your weights instead.
* Almost always works faster than the standard gradient descent algorithm.
  + Standard gradient descent is more susceptible to oscillations as you increase the learning rate (though feature scaling helps reduce these effects), and having a smaller learning rate requires more iterations. Example of oscillations in a contour diagram of the cost function .



* + Gradient descent with momentum effectively smooths out the steps. This is because the ceiling and floor of each oscillation will cancel each other, and so the general trend would take much more dominant role.
* Algorithm:

// matrix of zeros with same dimensions as and , respectively

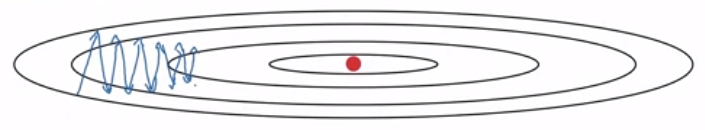
For each iteration:

Compute and on the current mini-batch.

* Now we have two hyperparameters to train: and .
  + Most common value of , roughly averaging over the previous 10 gradients.
* Some people use and instead of multiplying the derivative terms by .
  + This is fine as long as becomes scaled by .
  + In this version, any change the parameter will affect the scaling of and , which will need a change to . Hence, this version is not as preferred.

RMSprop

* RMSprop = root mean square prop
  + We take the square of derivatives, find the exponentially weighted average of the squared derivatives, and then take the square root.
* RMSprop achieves faster gradient descent by increasing the step size in the directions where we are further away from the global minimum and decreasing the step size in the directions we are closer to the global minimum.
  + For example, RMSprop will decrease the vertical component of the step size but increase the horizontal component of the step size for this contour diagram of the cost function:



* Algorithm:

// matrix of zeros with same dimensions as and , respectively

For each iteration :

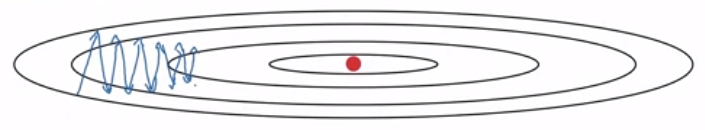
Compute on the current mini-batch.

// is an element-wise square operator.

// to handle case where

// to handle case where

* + RMSprop makes it so that the large is, the smaller the step is going to be. Similarly, the larger is, the smaller the step is going to take.
  + Going back to this example, the derivative for the vertical component is large (since the ovals are skinny in the vertical direction), so or would be large, which means we take a small step. The derivative for the horizontal component is small, so or would be small, which means we take a larger step.



Adam Optimization

* Adam = Adaptive Moment Estimation
  + and are the moments being adapted.
* Adam optimization puts together gradient descent with momentum and RMSprop together to get an even better gradient descent algorithm.
* Algorithm:

// matrix of zeros with same dimensions as

// matrix of zeros with same dimensions as

For each iteration :

Compute on the current mini-batch.

// momentum-like update with

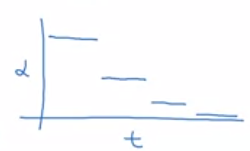
// RMSprop-like update with

// Bias correction is done by default in Adam optimization

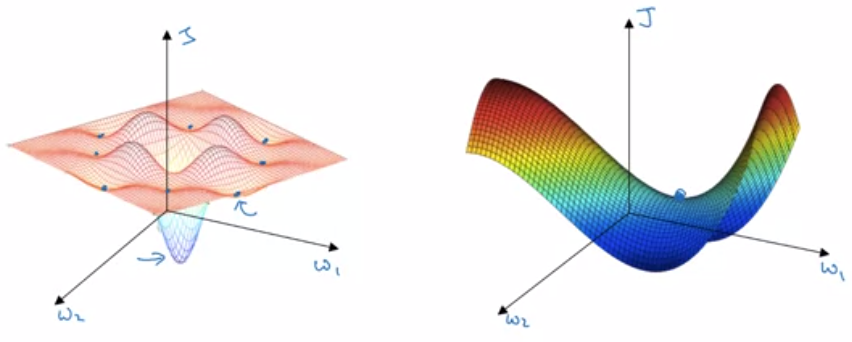
* This problem has been proven to be effective in a wide variety of neural network architectures.
* Hyperparameters choice:
  + needs to be tuned: just play around and experiment.
  + is usually 0.9
  + is usually 0.999
  + is usually (doesn’t affect performance much – don’t bother tuning)

Learning Rate Decay and The Problem of Local Optima

* Learning rate decay: slow your learning rate over time.
* Useful for mini-batch gradient descent when the batch size is small.
  + This is because mini-batch gradient descent never truly converges: it wanders around the global minimum instead.
  + Lowering the learning rate will cause the oscillations to decrease once the algorithm reaches the global minimum, thus causing it to actually converge.
  + Of course, we want to have the learning rate higher in the beginning to make larger steps. We only lower the learning rate after some iterations.
* Recall that 1 epoch is 1 pass through the training set.
* The algorithm:
  + Where is some initial learning rate, and is another hyperparameter.
  + Example: if and , then for the first epoch, . One the second epoch . On the third epoch . On the fourth epoch . Etc.
* Other learning rate decay methods:
  + Exponential decay: , for some (e.g. )
  + or (where is the mini-batch number)
  + A learning rate that decreases discretely like a staircase. Example:



* + Manual decay: if your model takes a long time to train (e.g. days), then manually reduce the learning rate after a period of time (e.g. a few hours).
* Learning rate decay creates more hyperparameters (including the learning rate decay method), but is lower on the list of priorities.
* In the early days of deep learning, people used to worry a lot about the optimization algorithm getting stuck in bad local optima.
  + However, if we have many parameters/features, most points with zero derivative are saddle points instead of local optima. This is because when the derivative is zero, for each variable, the variable can be concave or convex at that point. Only if all variables are convex at that point do we get a local optima – if any of them are concave, then we have a saddle point.
  + In the picture below, left diagram shows local optima and right diagram shows saddle point:



* Plateaus pose another problem. Plateaus are regions where the derivative is near zero for a long time. This makes gradient descent slow, but Adam Optimization, Momentum, and RMSProp can help.

**Hyperparameter Tuning, Batch Normalization, and Programming Frameworks**

Hyperparameter Tuning

* List of hyperparameters:
  + Learning rate
  + Momentum term or Adam Optimization algorithm parameters
  + Neural network architecture: # layers, # of hidden units
  + Learning rate decay
  + Mini-batch size
* Importance of hyperparameters
  + is the most important
  + , mini-batch size, # of hidden units are second in importance. is a good place to start.
  + # of layers and learning rate decay are third in importance
  + generally don’t need to be tuned if you’re using Adam Optimization algorithm. almost always work well.
* Selecting values to explore for the hyperparameters
  + In early machine learning days, people tried tuning hyperparameters in a grid fashion. For example (assume all hidden layers have same number of nodes):

For :

For

Train and test model with these hyperparameters.

* + However, it is better to try sampling random points. This would look something like (assume all hidden layers have same number of nodes):

For :

random value between 2 and 10.

random value between 2 and 40.

Train the test model with these hyperparameters.

* + The sampling-based approach is better is because it’s difficult to know which hyperparameters are more important in advance. This maximizes the amount of diverse hyperparameter values tested. (If you’re using the grid-based approach and one of the hyperparameters is relatively insignificant compared to another hyperparameter, you’re wasting time trying hyperparameter values where only the insignificant hyperparameter is being changed.)
* If you find a region of hyperparameter values that work well, try sampling more finely in that smaller region. This is referred to as coarse to fine search.
* Some parameters should be sampled uniformly, but some parameters should be sampled at a different scale
  + # of hidden layers and # of hidden units per layer can be sampled uniformly.
  + should be sampled logarithmically. One valid approach, for example, would picking a value between and should be the same likelihood as picking a value between and , a value between to , and a value between and . This can be implemented via Python code:

// pick random value between and

np.random.rand()

* + Hyperparameters for exponentially weighted averages : as approaches 1, the sensitivity of this value increases. For example, if increases from 0.9000 to 0.9005, nothing significant will change as both are an average of roughly the previous 10 values, but if increases from 0.9990 to 0.9995, that is significant since the former takes an average of roughly 1000 values and the latter takes an average of roughly 2000 values (using the formula ). Hence, we want to sample logarithmically. One valid approach would be to make take values between 0.1 and 0.001, where the chance of selecting a value 0.1 to 0.01 equal to the chance of selecting a value between 0.01 and 0.001. Algorithm:
* If you’re not sure what scale to use, then sampling uniformly and a coarse to fine search would generally give you ok results.
* People from different application domains do increasingly use ideas from other application areas to look for inspiration for cross-fertilization.
  + However, for hyperparameters, intuitions do get stale. As your data evolves over time (e.g. your data gradually changes over several months), your original best setting of hyperparameters may get stale. Hence, you should re-evaluate the hyperparameters you have occasionally.
* Two major schools of thought for searching for hyperparameters
  + Babysitting one model: this is particularly good if you have less computational resources. Manually change hyperparameter values as you train the model, assuming the model takes many hours/days to train, and observe how those changes affect the iteration # vs. curve. We refer to this as the “Panda” strategy since Pandas have few offspring, and pay attention to each one of them greatly.
  + Training many models in parallel: good if you have more computational resources. Separately train several versions of the model, where each version has a different set of hyperparameters compared to another version. We refer to this as the “Caviar” strategy since fish having many offspring; they don’t pay much attention to any specific one but rather see if a few do well.
* The Panda approach is good for applications where there may be a lot of data relative to the amount of computational resources to process that data.

Batch Normalization

* This approach may (but not always) make the neural network much more robust to the choice of hyperparameters (i.e. there’s a much bigger range of acceptable values), making the search easier and training go faster.
* In logistic regression, recall that we can normalize the inputs to make training and faster: . For neural networks, we can apply a similar idea of normalizing the inputs of each neuron to make training the weights associated with that neuron faster. The inputs of each neuron are the activations of the previous layer, .
  + In other words, can we normalize the values of to train and faster?
  + Note that we actually will choose to normalize instead of , though there are some debates in literature which should actually be normalized. In practice, normalizing is done more often.
* Implementing batch normalization: given some intermediate value in the neural network , we compute the normalized version of it and then :

For each mini-batch:

* + Ideally, we would use the mean and standard deviation across all training examples, but that can be inefficient to compute. Hence, we compute the mean and standard deviation for each mini-batch instead.
  + and are learnable parameters of the model. Note that and are vectors of dimensions .
  + Use instead of for later computations of the neural network.
  + The reason why we use instead of is to allow for different distributions (means and variances) to be learned in the hidden layers. This is different than normalizing the inputs, where we always want the input features to have mean 0 and standard deviation 1.
  + Note that if and , then we get the original distribution of (i.e. ).
* Adding batch normalization to the forward propagation of neural network:

For layer :

For each mini-batch:

Compute given // as we have done before

Compute given // using algorithm above

Compute given // where

* Adding batch normalization to the back propagation of neural network:
  + If using gradient descent, then compute and then update . Similarly, compute and then update .
  + Many deep learning programming frameworks (e.g. tensorflow) often don’t require you to implement batch normalization.
* Note that no longer is a parameter of the neural network if we’re using batch normalization, as has no effect on the neural network. This is because we subtract off from , where , so any constant added to across all examples (i.e. ) will not have any effect.
  + Instead, will carry a similar role to without batch normalization, as it shifts all the .
* A covariate shift is when the input distribution changes.
  + For example, you would like to train a neural network to identify cats but you provide it only examples of black cats, you wouldn’t expect it to predict as well for cats of other colors due to a covariate shift.
  + After learning a mapping from input to output, you may need to retrain the learning algorithm if there is a covariate shift.
* Batch normalization helps to overcome covariate shifts for a given hidden layer.
  + The inputs to any hidden layer change over each iteration, causing a covariate shift (change in input distribution) for that hidden layer.
  + Batch normalization reduces the amount that the distribution of the hidden units shift around, thus reducing covariate shift of the next layer of hidden units. It makes sure the mean and variance are the same among , which is governed by and .
* Batch normalization also has a regularization effect.
  + Each mini-batch is scaled by the mean/variance computed on just that mini-batch as opposed to the mean/variance across the whole training set. Since we compute mean/variance on a smaller set, there is more noise to the mean/variance.
  + This noise adds noise to the values of and thus within that minibatch. Similar to how dropout achieves regularization by adding noise, batch normalization adds some noise to each hidden layer’s activation and thus has a slight regularization effect.
  + Note that the regularization effect for batch norm is much less than dropout, and regularization is generally not the intent of batch normalization.
  + Using a larger mini-batch size reduces the noise of the mean/variance, which thus reduces the amount of regularization.
* For batch normalization, and are computed across the entire mini-batch during training, but there isn’t a mini-batch during testing. For example, the examples may come in one at a time, so the rules for calculating don’t make sense.
  + Thus, we instead estimate and using an exponentially weighted average for each layer across the mini-batches while training. More concretely, we set to be the exponentially weighted average of , etc. in that order, where is the average of the nodes in layer in the ’th mini-batch. Similarly, we set to be the exponentially weighted average of
  + We then use these estimates and during test time to compute .
  + and are still applied as normally, which were learned during the training process.
* In practice, any reasonable way to estimate and during test time is fine, as this approach is pretty robust.

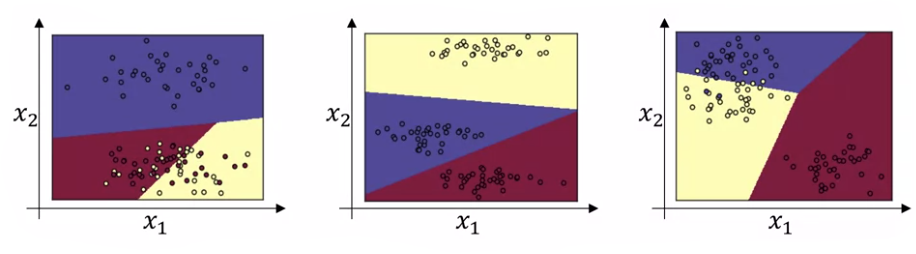
Multiclass Classification

* Notation: is the number of classes
  + Let us define classes as class 1, 2, 3, …, and let us define 1 class as class 0 for “none of the above”. Ex: class 1 = dog, class 2 = cat, class 3 = baby chick, class 0 = none of the above or other
* The output layer has units now, i.e.
  + An output node outputs the probability that the example belongs to the class, i.e. .
  + is a x 1 vector.
* The Softmax regression generalizes logistic regression (and thus neural networks) to classes. The Softmax layer ensures that the entries of sum to 1:
  + For the last layer, perform the following operations to get :

// is a 1 vector

// Take for all

* + We can summarize the above calculations with , where represents the Softmax activation function.
* Here are some examples of the decision boundaries that a Softmax activation function can represent. Note that the decision boundary between any two classes is linear.



* Softmax is in contrast to hardmax, which returns a vector where one entry is “1” and the other entries are 0. It simply selects “1” for the entry whose value is the largest in and set all other entries to 0.
* Softmax regression functions the same way as logistic regression when
* Loss function:
  + Since the ground truth is a vector containing a single 1 and remaining entries 0, is going to be 0 except for the entry where , from which is the loss of the training example. To minimize the loss function, we need to make as small as possible, or make as large as possible (though it is bounded by 1 of course).
  + This can be derived from the principle of maximum likelihood estimation.
* Cost function across all examples: .
* Vectorizing across all examples: is now a matrix and
* Backpropagation with softmax: , and continue as normally.
* Some programming frameworks (e.g. Tensorflow) can automatically perform the backpropagation steps if you provide it the forward propagation steps.

Introduction to programming frameworks

* As you implement very large models, it is increasingly impractical to implement everything yourself from scratch, so there are good deep learning software frameworks to help implement these models.
* Some of the leading deep learning frameworks: Caffe/Caffe2, CNTK, DL4J, Keras, Lasagne, mxnet, PaddlePaddle, TensorFlow, Theano, Torch
* Criteria for choosing deep learning frameworks:
  + Ease of programming (development and deployment)
  + Running speed
  + Truly open (open source with good governance)