**Bias/Variance**

* High bias = underfitting, high variance = overfitting
  + High training set error -> high bias
  + Low training set error, high test set error -> high variance
  + Not necessarily mutually exclusive
  + Should be compared to underlying base rate – how would humans fare on the same task? E.g. if a problem is already difficult for humans and accuracy is low, computers may not be able to perform much better
* Try to resolve high bias first, then high variance
* Ways to resolve high bias – larger network (# layers, # units), train longer, possibly – different architecture
* Ways to resolve high variance – More data, regularization, possibly – different architecture

**Train/Dev/Test Split**

* Not necc. to be 60/20/20 – purpose of dev(validation) set is to compare hyper parameters, and purpose of test set is to give unbiased estimate of error. With a very large dataset (e.g. 1 million +), may not need such a large size for testing/validating, e.g. may only need 10 thousand examples for test and 10 thousand for val, in which case, it would be a 98/1/1 split.
* Dev and test set should be drawn from the same distribution (sometimes train data generated from extra mining etc. in order to increase training set) for effective evaluation.

**Regularization Techniques**

* L2 norm regularization: add sum of square of weights to the cost
* L1 norm: add sum of absolute value of weights to the cost (LASSO -> makes some weights go to 0 faster)
* Dropout: (Modern implementation is inverted dropout). For each layer, set a keep-probability. During every iteration, randomly keep and drop hidden units in each layer based on that layer’s keep-probability, and then do the propagations, dividing the activations by the keep-probability (so that expectation value from the activations is not affected, ie weights won’t get correspondingly bigger) (Inverted dropout). No dropout during prediction.
* **Early Stopping**: algorithm to stop training of model midway, when training and test set errors begin to diverge.
  + Advantage – less computationally expensive than having to test multiple values of lambda for L2 regularization
  + Disadvantage – Mixes the problem of optimization together with not overfitting, making it harder to implement certain algorithms/think about certain effects
* **Data Augmentation**: Increasing a sparse data set by using current data set. E.g. for computer vision, instead of getting new images, can flip/randomly crop and rotate/slightly distort image, to obtain a new example in data set. Not as good as getting a completely new example, but trains the model that slight rotations/distortions etc. do not affect the labels

**Feature Scaling**

* Scale data by (x - mean)/stddev for faster optimization, use same scaling for test and train, and predictions

**Vanishing/Exploding gradients**

* For very deep neural networks, if (sum of) weights are initialized to be greater than 1 or smaller than 1 in each layer, will cause z to grow/decrease exponentially -> will also result in dz to grow/decrease exponentially, resulting in poor optimization
* Can slightly offset it by initializing (sum of) weights in each layer to be close to 1 – by dividing the random initialization by the variance
  + For Relu, initialize with W[l] = np.random.randn(shape) \* np.sqrt(2/n[l-1])
  + For tanh, initialize with W[l] = np.random.randn(shape) \* np.sqrt(1/n[l-1])
    - This is Xavier initialization
  + Some others may require factor of sqrt(2/(n[l-1]+n[l])) instead

**Gradient Checking**

* Numerically compute gradients to ensure there’s no bugs (ie [J(theta+epsilon) – J(theta-epsilon)]/(2\*epsilon)), where theta is all the weights and biases rolled out, and epsilon is very small
* Compare numerically calculated and backprop gradients using (||dthetaapprox – dtheta||2)/(||dthetaapprox||2 + ||dtheta||2) where ||x||2 indicates the sqrt L2-norm (ie the mathematical length of the vector)
* For epsilon = 10^-7, should obtain a result of 10^-7 or smaller. If result is 10^-5, be slightly worried, anything of the order 10^-3 and greater suggests a bug – compare each individual value and see whether the bug may arise from (e.g. computing dbias for a certain layer)
* Don’t use in training, only to debug (computationally expensive)
* Remember to include regularization term when calculating numerical gradient
* Doesn’t work with dropout
* Sometimes works for small w, b, but not for larger ones. May want to check at random initialization, and again after some training.

**Mini-batch Size**

* If small training set (m <= 2000), just use batch gradient descent
* Otherwise, usually from 64-512 (sometimes 1024):
  + Try to use a power of 2 – usually faster due to how computers are implemented
  + Make sure the minibatch fits into CPU/GPU memory to make use of its advantages

**Hyperparameter Tuning**

* By importance, the hyperparameters one should tune are:
  + Most important: Learning Rate
  + Next: Momentum parameter (if using), no. hidden units, mini-batch size
  + Third: No. of layers, learning rate decay
  + Least: Adam hyperparameters (beta 1, beta 2, epsilon)
* When selecting a range of choices of hyperparameters to test:
  + Don’t select fixed values for each hyperparameter and then go systematically – if some hyperparameters have not much effect, then essentially only have a few points for the hyperparameters that matter
  + Select using random values, so that there’s more data points for each hyperparameter
    - For no. of layers/no. of units, uniform distribution over the range may be alright
    - For learning rate, etc., with large changes in magnitude, will want to sample uniformly on a log scale – i.e. select randomly from a uniform distribution for exponents, then take 10^exponent to get the random value
    - For hyperparameters tuning exponentially weighted averages (e.g. beta in momentum), use the same uniform sampling process on a log scale for 1 - beta
* Coarse to fine – start with a larger search range, then can close in on smaller ranges for each hyperparameter and search again
* Try relooking at hyperparameters once every few months – changes in architecture/servers etc. could mean that optimal hyperparameters has changed
* If many computational resources, can train multiple models simultaneously to test hyperparameters. Otherwise, if training takes a long time and not enough computational resources to run multiple models, can tune hyperparameters daily before resuming training.

**Avoidable Bias**

* How to tell how much bias is in the system? Problem may just be difficult
* Try to estimate/evaluate the human error rate – that should be initial target of model to achieve. Difference between human level and training error will be the avoidable bias (the bias that can definitely be improved upon by a better model)
* True Bayes error rate (the smallest error rate possible by anything – since any task is also subject to random noise/extreme difficulties that makes the task impossible – e.g. extremely blurry photo where it is impossible to figure out if it is of a cat or a dog) may be lower than human error rate (but is never higher, by definition), in which case, ML may be able to perform better than human error rate
* In this regime, it is difficult to tell how much more avoidable bias there is, and may be harder to tell if bias or variance is the bigger problem

**Error Analysis**

* Manually look at mislabelled things in dev set – if there’s a significant type of thing it usually gets wrong/a common issue across the mislabellings, can consider focusing on improving on that issue to improve on model
* Sometimes issue may be due to incorrectly labelled data (by the human, i.e. y\_true is not correct) – if it is a significant enough issue that affects selection of the model (error percentage is a close to or larger than error percentage difference between models), then it may be worth it to relabel them
  + May be worth it to look at mislabels in the ‘correctly predicted’ things too

**Data Mismatch**

* If training model for a specific task where the specific dataset is not very large, sometimes may augment with a larger, more general data set with a different distribution for training (e.g. training to recognize images taken by users of an app, and then augmenting the training set with professionally shot photos from the web)
* Test and Dev set should reflect goal – only use the specific dataset for test and dev, and then use some for training. Don’t use general dataset in test/dev as it wont accurately reflect goal – will end up fitting to larger general dataset
* Now – if only have training error, dev error, and test set error, may not be able to tell if there’s a high variance problem, or if dev/test dataset is just a more difficult problem. Hence, create a training-dev set as well, which is drawn from the ‘training’ dataset. Train data on training set, then evaluate on training-dev set, and dev set. Comparing training set error and training-dev set error shows the variance, comparing training-dev set and dev set error shows if there is a data mismatch problem
* If there is a data mismatch problem, can try synthesizing artificial data to make training set distribution more similar to dev/test set distribution – e.g. if dev/test set is noisier/blurrier in general, can alter the training data in such a way first before training
  + However, must be careful of how this additional step is done – to a human, maybe noise all sounds the same, but e.g. if only have 1hr of noise and 10,000hrs of training data which noise is applied to, the model may end up overfitting to that 1hr of noise.
  + Or maybe using CGI images to augment image dataset (e.g. taking pictures of cars from a video game) – however, the video game may have only ~40 kinds of cars, which looks like a wide variety to a human, but not a computer, where it has to learn to recognize cars from a much larger variety – hence, overfit to a subset again