Determine goals - what error metric to use and how low to get this metric
Establish a working pipeline (system) as soon as possible
Make it easy to diagnose issues in the model
Make incremental changes such as increasing number of neurons in a layer

Performance Metrics
Bayes error rate defines the minimum error rate you can hope to achieve
Design decisions should be based on reaching a goal error rate metric
Performance metrics are different from the cost function

Sometimes it is good to train a binary classifier to detect a rare event (e.g 1 in a million cancer)
If the classifier always predicts no cancer, it is right 99.9999% of the time - accuracy is bad
Precision is the fraction of detections by the model which were correct, while recall is the fraction of true events which were detected

It is common to plot a PR curve with precision on the y-axis and recall on the x-axis
It is common to report a detection whenever the classifier's output exceeds a threshold
By varying the threshold, we can trade precision for recall

It is also possible to use one number instead of a curve to summarize things; this is the F-score

$$F = \frac{2pr}{p+r}$$

Another option is the area under the PR curve

If a model is not confident in its predictions, then it can choose not to output anything - this is good when it is important to not make a mistake
Coverage is the fraction of training examples for which the model is able to produce a response

Default Baseline Models
Identify the general architecture (e.g image = convnet)
Apply regularization, dropout, and batch normalization
If unsupervised learning is important, include it in your first end-to-end baseline

Determining Whether to Gather More Data
If the performance on the current training data is poor, it is usually a result of data quality (e.g noise), model capacity, or model hyperparameters
If the test set performance is much worse than the training set, it is a good idea to get more data

It is recommended to plot a experiment on a logarithmic scale of the curve of training data vs. generalization error, and then to extrapolate the necessary training data from there

Selecting Hyperparameters

Automatic hyperparameter selection makes things less tedious but is much more computationally costly
Manual hyperparameter selection attempts to adjust the effective capacity of the model to match the complexity of the task - e.g setting the weight decay coefficient such that the end learned result is the correct function

Hyperparameter selection vs error is a U-shaped curve with one extreme being underfitting and the other being overfitting
If your training set error is higher than your target error and the model is otherwise perfect, the only way to fix this is to increase the capacity (add more layers)

The goal is to reduce the gap between training and test error without increasing training error
This is best done through dropout or weight decay

The brute force method of decreasing test error is to increase model capacity and training set size until the error is sufficient

Automatic Hyperparameter Optimization Algorithms
Hyperparameter optimization has its own hyperparameters to choose, such as the range of values which should be explored
However, this is a much easier secondary task compared to choosing hyperparameters of a neural net

In grid search, for each hyperparameter, the user selects a small set of values to explore
The grid search algorithm trains a model for every set of hyperparameters in the Cartesian product (all combinations of each x and y) of the set of values, with each dimension in the Cartesian product given by the hyperparameters

Select the lower and upper limits conservatively based on prior experiments

An example of a logarithmic scale is $\{.1, .01, 10^{-3}, 10^{-4}, 10^{-5}\}$
Keep on zooming in / conducting refined grid searches based on previous grid searches
Grid search has an exponential cost of n^m with m hyperparameters taking on at most n values

- Can be parallel computed

For a random search, define a marginal distribution for each hyperparameter (e.g Bernoulli for binary) or a uniform distribution on a log-scale for positive real-valued hyperparameters

$$\texttt{log\_learning\_rate} \sim u(-1, -5)$$
$$\texttt{learning\_rate} = 10^{\texttt{log\_learning\_rate}}$$

where u(a, b) indicates a uniform distribution in the interval (a, b)

Unlike in grid search, one should not discretize or bin the values of hyperparameters in random search - when there are hyperparameters which do not significantly affect the performance measure, random search is much faster

Random search tends to be more efficient since grid search changes one parameter at a time while the others are constant, so it essentially repeats trials if a hyperparameter is not useful
Random search changes all hyperparameters independently of one another, so it has new experiments

When it is feasible to compute the gradient of a differentiable error on the validation set w.r.t the hyperparameters, we can simply follow the gradient
This is not possible in most cases however, since the hyperparameter is usually discrete or this just has too high a computational cost

To compensate for the lack of a gradient, we can build a model of the validation error and propose hyperparameter guesses by optimizating within this model (although this model probably will not be precise)
Most model-based algorithms use Bayesian regression to estimate the expected value of the validation set error for each hyperparameter along with an uncertainty
Optimization involves a tradeoff between exploration and exploitation (using known hyperparams)

Debugging Strategies
Machine learning models have components which are adaptive
Suppose we made an error in gradient descent for biases:
$b \leftarrow b - \alpha$ where **α** is the learning rate
The weights may learn to adapt to these poor biases

We can design a case which is so simple that the correct behavior can be predicted, or we design a test which exercises part of the neural net implementation in isolation

Visualizing the model in action can help determine whether it is doing what it needs to - metrics aren't always truthful
Most models are able to output some sort of confidence measure for each task they perform
The probability assigned to the most likely class gives an estimate of the confidence the model has in its classification decision
Max likelihood training typically results in overestimates rather than accurate probabilities of the correct prediction, but the predictions are still useful since examples which are less likely to be correctly labeled receive smaller probabilities in the model

By viewing the training set examples which are hardest to model correctly, one can see how there are problems with the way the data is labeled or preprocessed

Choosing to display the most confident mistakes on images can reveal flaws in the data itself

Test error may be high if the testset was prepared differently than the training set

Usually, if you cannot train a classifier to correctly label a single example, or train an autoencoder to reproduce an example, or a generative model to create a slight resemblance, there is a software defect preventing successful implementation

If you are making your own bprop for a certain operation, or implementing a numerical derivative on your own, implementing the gradient expression incorrectly is a common source of error

We can verify that these derivatives are correct by comparing our implementation of autodiff to the derivatives computed by finite differences

Since $$f'(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon},$$, we can approximate this using a small finite ϵ

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}.$$

We can improve accuracy by using a centered difference

$$f'(x) \approx \frac{f(x+\frac{1}{2}\epsilon) - f(x-\frac{1}{2}\epsilon)}{\epsilon}.$$

ϵ must be large enough such that finite precision does not affect the computation

We want to test the gradient or Jacobian of a vector-valued function g : R^m -> R^n
We can run finite differencing mn times to evaluate all partials of g, or we can apply the test to a new function which projects the inputs and outputs down

We can apply our test of the implementation of derivatives to f where f(x) = $\boldsymbol{u}^T g(\boldsymbol{v}x)$, where u and v are randomly chosen vectors
Computing f' requires being able to backpropagate through g correctly - we are still using the computational graph of g to backpropagate
It is efficient to do so with finite differences since f only has one input and one output (whereas g has multiple)
Repeating this tests w/ varying u and v is recommended to avoid accidentally ignoring components orthogonal to the projection

There is a very efficient way to estimate the gradient using complex numbers

$$f(x+i\epsilon) = f(x) + i\epsilon f'(x) + O(\epsilon^2)$$

$$\text{real}(f(x+i\epsilon)) = f(x) + O(\epsilon^2), \quad \text{imag}(\frac{f(x+i\epsilon)}{\epsilon}) = f'(x) + O(\epsilon^2).$$

There is no cancellation effect due to taking the difference between f at different points (finite precision is not an issue), allowing the use of tiny values ϵ at 10^{-150} making $O(\epsilon^2)$ insignificant

It is often useful to visualize the stats of a neural net's activations and gradients
The pre-activation value of hidden units tell us if the units saturate (e.g for tanh the average of the absolute value tells us if it saturates during training)
It is useful to compare the magnitude of the gradients to the magnitude of the parameters (gradients should not be 50% of the parameter's magnitude)

Many deep learning algorithms have a guarantee about the results they produce at each step - these may include the gradients being 0 at convergence, or the objective function not increasing

Example: Multi-Digit Number Recognition
Google Street view first collected raw data and used detection algorithms to detect addresses
Due to the high accuracy requirement, the main goal became to optimize coverage - the confidence threshold below which the network refused to transcribe the input was reduced as the convnet became better

There were n softmax units, each trained to predict a sequence of n characters in the address; they were trained independently

The initial definition of p(y|x) where p(y|x) was checked against a threshold was to multiply all the softmax units together
This encouraged a specialized output layer and cost function which computed a principled log-likelihood, making the network function much better

Since the training and test error were similar, the researchers checked for error in the training data, and found that images were too narrowly cropped

Each square maintains the game's state - to check for the winner, maintain the state of the 9 squares in one location