STA 4241 Lecture, Week 14

Ensemble approaches

- **Ensemble approaches**
    - Super learner
- Introduction to neural networks
    - Projection pursuit regression
    - Hidden layer neural network
- When to use neural networks and why do they work?

Ensemble approaches

- Today, we will discuss ensemble learning methods.
- Ensemble learning is an approach for combining many different prediction models.
- We've already seen some ensemble learning methods
  - Bagging
  - Boosting
  - Random forests
- One particularly popular instance of ensemble learning is known as *super learning*.

Bagging

- Recall the bagging (bootstrap aggregated) estimator we discussed earlier in the course.
- To fit the bagging estimator, we constructed M bootstrapped datasets from our full training data $\{(Y_i, X_i)\}_{i=1}^n$
- Then, we fit a prediction model to each of the bootstrapped datasets, denoted by $\hat{f}_m(x)$
- The bagging estimator was then

$$\hat{f}_{\text{bag}}(X) = \frac{1}{M} \sum_{m=1}^{M} \hat{f}_m(X).$$

- The utility of bagging was that it could reduce the variance by averaging.

Bagging as model averaging

- We can alternatively think of $\hat{f}_{\mathrm{bag}}$ as a model averaging approach: after all, it is defined as simply the average of M different prediction models $\hat{f}_m$.
- From this perspective, we can view model aggregation as a slightly different problem: given models $\{\hat{f}_1, \ldots, \hat{f}_M\}$, how can we combine them in order to achieve the best combination?
    - Best in terms of MSE or classification error
- We could take the average, but maybe a weighted average is preferable

Model averaging

- Let $\hat{f}_1(X), \ldots, \hat{f}_M(X)$ be predictions at X coming from M distinct fitted models.
- Assume for now we have a continuous outcome and reducing MSE is our goal
    - Straightforward extensions to categorical outcomes
- Given these predictions we seek weights $w = (w_1, \ldots, w_M)$ defined as

$$\arg\min_{w} \mathbb{E}\left[\left\{Y - \sum_{m=1}^{M} w_m \hat{f}_m(X)\right\}^2\right]$$

- Expectation is taken with respect to the distribution of Y given X

Model averaging

- Our prediction is now defined as

$$\widehat{Y} = \sum_{m=1}^{M} w_m \hat{f}_m(X)$$

- And our goal is to find weights w that make $\widehat{Y}$ as close to Y as possible
- In bagging we assigned weights of $w = (1/M, 1/M \ldots, 1/M)$
- Now we want to find the optimal weights to reduce MSE

Model averaging

- What do we want our weights to satisfy?
  - Small weight for any prediction with a large bias
  - Small weight to predictors with large variance
  - Larger weight to predictors that are more independent of the other predictors
- Suppose we have 5 highly correlated predictions and 5 independent ones
  - Assign small weight to the correlated ones, and high weight to the independent ones
  - This minimizes the variance of the model averaged prediction

Model averaging

- It turns out the optimal weights have a simple form
- Let $\hat{F}(X) \equiv [\hat{f}_1(X), \ldots, \hat{f}_M(X)]$
- We can show that the optimal weights are given by

$$\hat{w} = \mathbb{E}\left[\hat{F}(X)\hat{F}(X)^T\right]^{-1}\mathbb{E}\left[\hat{F}(X)Y\right].$$

- The first term takes into account the covariance of the predictions while the second accounts for how close they are to Y

Model averaging

- In some sense this is very powerful
- There is a result that if we use these weights, then

$$\mathbb{E}\left[\left\{Y - \sum_{m=1}^{M} \hat{w}_m \hat{f}_m(X)\right\}^2\right] \leq \mathbb{E}\left[\left\{Y - \hat{f}_m(X)\right\}^2\right], \quad \forall m,$$

- This means that the weighted average model has as good or better MSE than any individual prediction model
- Is this surprising?
  - The individual models are simply special cases of the averaged one where the weights are $w = (0, \ldots, 0, 1, 0, \ldots, 0)$

Model averaging

- This seems great! We can always beat any individual model
- Unfortunately, these expectations must be estimated

$$\mathbb{E}\left[\hat{F}(X)\hat{F}(X)^T\right], \qquad \mathbb{E}\left[\hat{F}(X)Y\right]$$

- Replacing them with sample estimates can easily lead to poor prediction performance
  - Will place too much emphasis on overly complex prediction models
  - Overly complex models have low training error and appear to have good fits
  - Doesn't generalize to new data points well

Model averaging

- What have we seen throughout class that helps avoid overfitting
    - Cross-validation
- We can use cross-validation to help us choose weights that minimize prediction error on new data points
- This was first introduced in an approach called stacking, but has been popularized by a generalization of this approach called super learner

Stacking

- One approach for model combination which avoids overfitting is *stacking* or *stacked generalizations*
- Let $\hat{f}_m^{-i}(X)$ be the prediction at X using model m, estimated on a dataset with the ith sample removed.
- The stacked estimate of the weights w is then defined as

$$\hat{w}^{st} = \arg\min_w \left[\sum_{i=1}^{n} \left\{ y_i - \sum_{m=1}^{M} w_m f_m^{-i}(X_i) \right\}^2 \right\}$$

so that the stacked estimate at X is $\sum_{m=1}^{M} \hat{w}_m^{st} \hat{f}_m(X)$ where $\hat{f}_m$ is the mth model fit to the entire dataset.

Stacking

- This approach can be improved by imposing a non-negativity and sum-to-one constraint on the w

$$\tilde{w}^{st} = \arg\min_{w} \left[\sum_{i=1}^{n} \left\{ y_i - \sum_{m=1}^{M} w_m f_m^{-i}(X_i) \right\}^2 \right],$$

$$\sum_{m=1}^{M} w_m = 1, \quad w_m \geq 0 \quad \forall m.$$

By using cross-validation predictions $\hat{f}_m^{-i}(X_i)$, stacking avoids giving unfairly high weight to models with higher complexity.

Super learner

- One approach for model combination, which is a popular variation of generalized stacking, is *super learning*.
- The prediction model output at the end of the procedure is the so-called *super learner*.
- This method has achieved widespread use
    - Nice theoretical results
    - Easy (relatively) to use software

Super learner

- Suppose we have observed n realizations of the random pair (Y, X)
- The goal is to estimate $\mathbb{E}(Y \mid X) \equiv f(X)$
- For a particular dataset, suppose we have a library of estimators of f
    - These can be very simple or highly complex
- Each $\hat{f}_m$ needs to take in X and produce a prediction $\widehat{Y}$
- Let $\mathcal{L}$ denote the library of estimators and let M denote the cardinality of $\mathcal{L}$.

Algorithm for super learner

1. Fit each estimator in $\mathcal{L}$ to the entire training dataset to obtain $\hat{f}_1(X), \cdots \hat{f}_M(X)$.

2. Split the training data into a training and validation sample according to a V-fold cross-validation scheme. Define $T(\nu)$ to be the training data and $V(\nu)$ to be the validation data for the νth split for $\nu = 1, \ldots, V$.

3. For the νth fold, fit each algorithm in $\mathcal{L}$ on the training data $T(\nu)$. Store the predictions on the corresponding validation data $V(\nu)$, say, $\hat{f}_{m,T(\nu)}(X_i), X_i \in V(\nu)$. Repeat for $\nu = 1, \ldots, V$.

Algorithm for super learner

4. Stack the predictions from each estimator together to create an $n \times M$ matrix Z
   - The (i, m)th entry is the prediction for the ith data point from the mth estimator
   - Importantly, this prediction comes from the model that did not include observation i
5. Propose a family of weighted combinations of the candidate estimators indexed by weight vector w, e.g.,

$$m(Z_i, w) = \sum_{m=1}^{M} w_m Z_{i,m}, \quad w_m \geq 0 \quad \forall m, \quad \sum_{m=1}^{M} w_m = 1,$$

for $i = 1, \dots, n$.

Algorithm for super learner

6. Determine the w that minimizes the cross-validated risk of the candidate estimator, e.g.,

$$\hat{w} = \arg\min_{w} \sum_{i=1}^{n} \{Y_i - m(Z_i, w)\}^2 \quad w_m \geq 0 \quad \forall m, \quad \sum_{m=1}^{M} w_m = 1.$$

7. Output the SuperLearner $f_{\mathrm{SL}}(X) = \sum_{m=1}^{M} \hat{w}_m \hat{f}_m(X)$
   - Note the final output uses the models $\hat{f}_m(X)$ from the full training data

Super learner

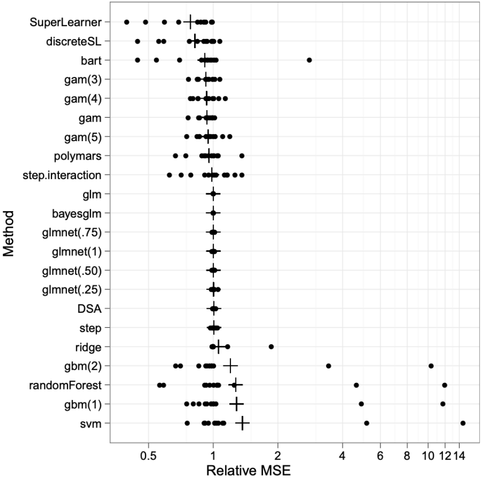
- The algorithm is quite flexible and performs quite well
  - $m(Z_i, w)$ does not need to be linear
  - Works immediately for classification problems or with loss functions other than MSE
- The main idea is fairly simple and intuitive
  - Averaging improves prediction when we have many candidate models
  - Use cross-validation to find weights that improve out of sample prediction

Super learner

- A very interesting result was proved about super learners, which is that asymptotically (as our sample size gets very big) super learners will perform as well or better than the best of the individual estimators
  - Does not require prior knowledge about individual estimators
  - Purely data driven
- There are no guarantees in finite samples, but it has been shown to work quite well for a wide range of problems
  - My personal experience is that it is hard to beat the super learner

Super learner

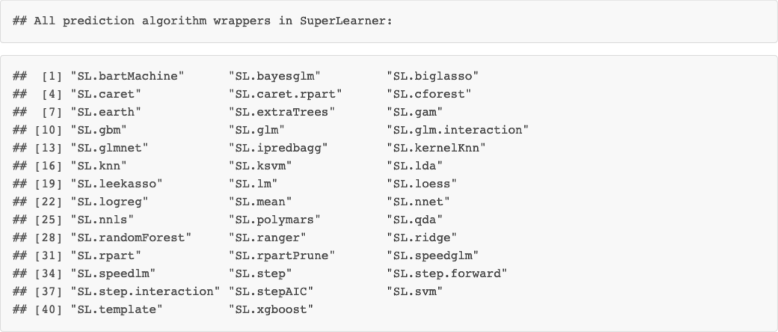
- Here is the predictive performance of super learner across multiple data sets



Van der Laan et al., 2007

Super learner

- The software package in R for super learners has a huge library of candidate estimators you can use
  - Many of these we have covered in class



```
## All prediction algorithm wrappers in SuperLearner:
```

```
##  [1] "SL.bartMachine"      "SL.bayesglm"         "SL.biglasso"
##  [4] "SL.caret"            "SL.caret.rpart"      "SL.cforest"
##  [7] "SL.earth"            "SL.extraTrees"       "SL.gam"
## [10] "SL.gbm"              "SL.glm"              "SL.glm.interaction"
## [13] "SL.glmnet"           "SL.ipredbagg"        "SL.kernelKnn"
## [16] "SL.knn"              "SL.ksvm"             "SL.lda"
## [19] "SL.leekasso"         "SL.lm"               "SL.loess"
## [22] "SL.logreg"           "SL.mean"             "SL.nnet"
## [25] "SL.nnls"             "SL.polymars"         "SL.qda"
## [28] "SL.randomForest"     "SL.ranger"           "SL.ridge"
## [31] "SL.rpart"            "SL.rpartPrune"       "SL.speedglm"
## [34] "SL.speedlm"          "SL.step"             "SL.step.forward"
## [37] "SL.step.interaction" "SL.stepAIC"          "SL.svm"
## [40] "SL.template"         "SL.xgboost"
```

Ensemble approaches

- Ensemble approaches
    - Super learner
- **Introduction to neural networks**
    - Projection pursuit regression
    - Hidden layer neural network
- When to use neural networks and why do they work?

Disclaimers

- I am not an expert in deep learning or neural networks!
- I have a working understanding of these concepts and thought it would be good for you all to be exposed to it due to its widespread use
- Don't worry if you don't understand everything that follows
    - These are very complex models
- We won't go into great technical detail here, but will instead give an overview of why these models work

Projection pursuit regression

- Before discussing neural networks, we discuss an approach called projection pursuit regression (PPR)
    - Many similarities to neural networks
    - Neural networks build on the ideas of projection pursuit regression
- The main idea of these approaches is to find new features, which are linear combinations of the original features
- Then run a highly nonlinear regression on these new features

Projection pursuit regression

- As is typically the case, our goal will be to estimate $\mathbb{E}(Y|X) = f(X)$
- The projection pursuit regression model is defined as

$$f(X) = \sum_{m=1}^{M} g_m(\omega_m^T X)$$

- This looks a lot like a generalized additive model
- Additive in the derived features, $V_m = \omega_m^T X$

Projection pursuit regression

- In this model there are two key unknown parameters to be estimated
  - The directions or weights given by ω_m
  - The nonlinear functions $g_m(\cdot)$
- The model simultaneously tries to find directions and functions that fit the data well
  - Most approaches we've considered have done only one of these at a time
  - Principle components regression is a special case where $g_m(x) = x$ and ω_m is chosen via PCA
  - Generalized additive models are the special case where $m = p$ and $\omega_m X = X_m$

Projection pursuit regression

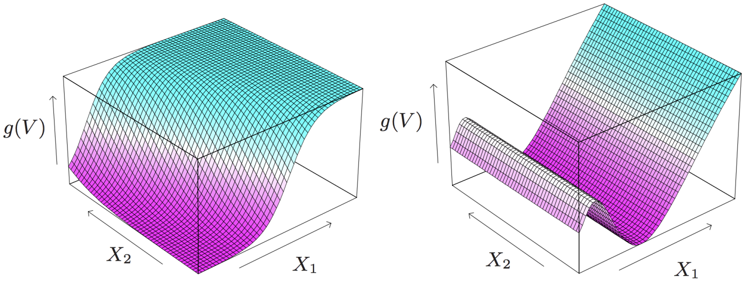
- This class of models is extremely large
  - Can account for high degrees of nonlinearity or non-additivity
  - Additivity in the derived features does not imply additivity of the original ones
- For instance, $X_1 X_2 = [(X_1 + X_2)^2 - (X_1 - X_2)^2]/4$
- Amazingly it can be shown that for a large enough M and correct functions g_m, **any** continuous function can be approximated arbitrarily well by the PPR model
  - PPR is a universal approximator

Projection pursuit regression

- This universal approximation property is quite incredible
    - Very flexible approach
- The main drawback is in interpretation
    - Not unique to PPR
    - Most machine learning approaches have this limitation
- PPR is best used for making predictions, not for inferring structural features of the model

Projection pursuit regression

- The $g_m(\cdot)$ functions are called ridge functions
- Below are two example ridge function / direction combinations when we have two covariates



Hastie, T., Tibshirani, R. and Friedman, J. (2009). The elements of statistical learning. New York: springer.

Projection pursuit regression

- So how do we estimate these complex models?
- We can aim to minimize the squared error

$$\sum_{i=1}^{n} \left\{ Y_i - \sum_{m=1}^{M} g_m(\omega_m^T X_i) \right\}^2$$

with respect to the functions g_m and vectors ω_m for $m = 1, \ldots, M$

- Need to impose some structure on the g_m functions as well
    - Avoid overfitting

Projection pursuit regression

- To simplify, let's first assume $M = 1$ so we only have one term
  - This is called a single index model and is widely used in some fields
- To simultaneously estimate g and ω we use an iterative procedure
  - First estimate g, then conditionally on that, estimate ω
  - Repeat until convergence
- Each of these two individual steps is relatively straightforward

Projection pursuit regression

- Suppose we know ω and want to estimate g
- Therefore we know $V = \omega^T X$ and we just need to estimate $g(V)$
- This is simply a one-dimensional smoothing problem!
  - See lectures 10 and 11 for a wide range of ways to solve those
- Smoothing splines and local regression are most convenient for computational reasons

Projection pursuit regression

- Now suppose we have estimated g and need to estimate ω conditional on g

- Letting $\omega_{(t)}$ be the t^{th} iterate of an algorithm to update ω with g fixed. We can approximate $g(\omega^T X_i)$ using

$$g(\omega^T X_i) \approx g(\omega_{(t)}^T X_i) + g'(\omega_{(t)}^T X_i)(\omega - \omega_{(t)})^T X_i$$

- Therefore we can write

$$\sum_{i=1}^{n} \Big\{ Y_i - g(\omega^T X_i) \Big\}^2 \approx$$

$$\sum_{i=1}^{n} g'(\omega_{(t)}^T X_i)^2 \left\{ \left(\omega_{(t)}^T X_i + \frac{Y_i - g(\omega_{(t)}^T X_i)}{g'(\omega_{(t)}^T X_i)} \right) - \omega^T X_i \right\}^2 .$$

Projection pursuit regression

- Interestingly this is simply a weighted least squares problem to solve for ω
  - The responses are $\omega_{(t)}^T X_i + \frac{Y_i - g(\omega_{(t)}^T X_i)}{g'(\omega_{(t)}^T X_i)}$
  - The inputs are the X_i's
  - The weights are $g'(\omega_{(t)}^T X_i)^2$
- Can use standard software for weighted least squares to solve this step
- Can iterate between this step and the previous step until convergence

Additional considerations

- This is how the problem is solved for $M = 1$
- When $M > 1$ we first fit the first function in this manner, then solve for (g_2, ω_2) conditionally on that first step to capture any additional signal, and so on
  - Forward stagewise model building
  - Similar to boosting
- The number of derived features M is an important tuning parameter
  - Stop when additional functions don't improve model fit substantially
  - Use cross-validation to find M

Difference between PPR and neural networks

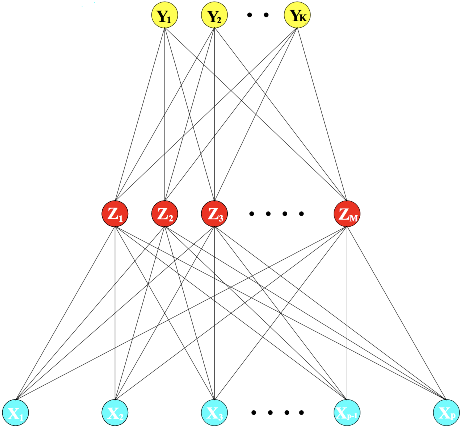
- Neural networks and PPR are very similar, but have a couple of key differences
- Neural networks do not estimate $g_m(\cdot)$ functions, but rather choose them a priori
- This would seem to be a restriction
  - Learning the functions adaptively should improve performance
- Neural networks use a much larger M typically
  - PPR usually restricts to M between 5 and 10
  - Neural networks will use much more

Neural networks

- For neural networks we need to adopt some new terminology
- We refer to our covariates as inputs, X_j
- The outputs are given by $Y_1, \ldots, Y_K$
  - For regression with a single response, we only have $K = 1$
  - For classification where the outcome has K classes, we have K outputs, each corresponding to a $0/1$ variable denoting membership into a particular class
- Derived features Z_m are called the hidden layers, because they are not directly observed
  - They are functions of our inputs

Neural networks

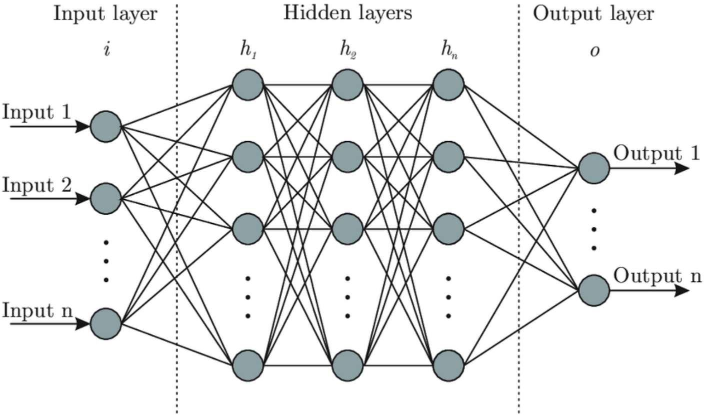
- This can be seen easily via a diagram
  - This represents a neural network with only one hidden layer



Hastie, T., Tibshirani, R. and Friedman, J. (2009). The elements of statistical learning. New York: springer.

Neural networks

- Can also have many hidden layers, but we will restrict attention mostly to the single hidden layer case



Neural networks

- Mathematically, we can write the single hidden layer model as follows

$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X), \quad m = 1, \dots, M$$

$$T_k = \beta_{0k} + \beta_k^T Z, \quad k = 1, \dots, K$$

$$f_k(X) = g_k(T), \quad k = 1, \dots, K$$

- This looks very similar to projection pursuit
  - We have replaced $g_m(\cdot)$ with a pre-specified function $\sigma(\cdot)$
  - $\sigma(\cdot)$ is called the activation function

Neural networks

- The most common choice for $\sigma(\cdot)$ is the sigmoid function

$$\sigma(v) = \frac{1}{1 + e^{-v}}$$

- Our outputs T_k are functions of our nonlinear derived features
- The final line of our model translates our outputs T_k to the scale of our outcome
  - For regression and continuous outcomes, we set $g_k(T_k) = T_k$
  - For classification, we can use the softmax function that gives us probabilities of class membership that sum to 1

$$g_k(T_k) = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}}$$

Neural networks

- This looks like a very fancy and complicated model, but ultimately all we have is a very nonlinear and non-additive regression model
- By setting M large, we can capture an extremely wide range of possible functions using this formulation
- Additional complexity can be added by including more hidden layers
  - Important tuning parameter
  - Many ways to add more layers
  - Fitting the best neural network is something of an art

Estimating neural networks

- There are a huge number of parameters we have introduced, which we can denote by θ
- These consist of

$$\{\alpha_{0m}, \alpha_m; m = 1, \dots, M\}; \qquad M(p+1) \text{ weights}$$

$$\{\beta_{0k}, \beta_k; k = 1, \dots, K\}; \qquad K(M+1) \text{ weights}$$

- For continuous outcomes we aim to minimize

$$R(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{n} \{Y_{ik} - f_k(X_i)\}^2$$

- Categorical outcomes can use cross-entropy (not covered here), but all other ideas apply directly

Estimating neural networks

- We will use a method called gradient descent to optimize this function
  - Or it's stochastic extension
- We don't want the value of θ that is the global minimizer of $R(\theta)$
  - This solution would be overfit
- Regularization is induced in one of two ways
  - Early stopping of the optimization algorithm (indirect penalty)
  - Applying a penalty term

Brief overview of gradient descent

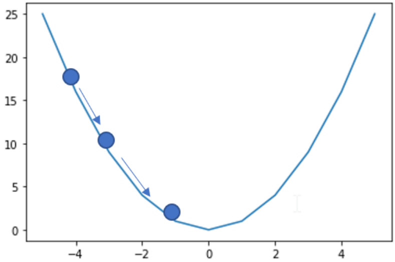
- Gradient descent is an iterative method for finding local minima or maxima of a differentiable function
- If we are currently at $\theta_{(t)}$ then we set

$$\theta_{(t+1)} = \theta_{(t)} - \gamma R'(\theta_{(t)})$$

- The main idea is to move θ in the opposite direction of the derivative as this will take you towards the minimum
- This is easiest to see visually in one dimension

Brief overview of gradient descent

- If we start to the left of the minimizing value the derivative is negative
- Therefore we move to the right to get closer to the minimizing value
  - Repeat this until convergence



Neural networks

- Neural networks use gradient descent to find the weights in the model
- In practice they use stochastic gradient descent which speeds up computation
    - Randomly select a subset of data points to use when calculating the gradient
- One form of penalization is to stop this algorithm before convergence
- This seems like an odd way to induce regularization, but in practice it does well at shrinking the model towards linearity
- In special cases there is an explicit connection between early stopping and ridge regression

Neural networks

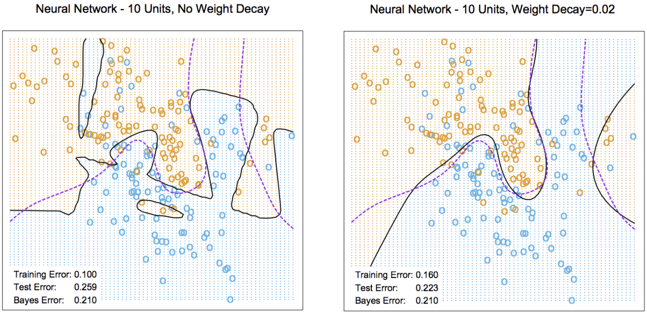
- Weight decay is another approach to regularizing neural networks
  - Similar to more standard penalization approaches
- Now we aim to minimize

$$R(\theta) + J(\theta) = R(\theta) + \sum_{k,m} \beta_{km}^2 + \sum_{m,l} \alpha_{ml}^2$$

- This places a ridge penalty on the weights and shrinks them towards zero
- The same gradient descent approaches apply with this penalty

Neural networks

- Here we see on an example that the weight decay can substantially improve model fit



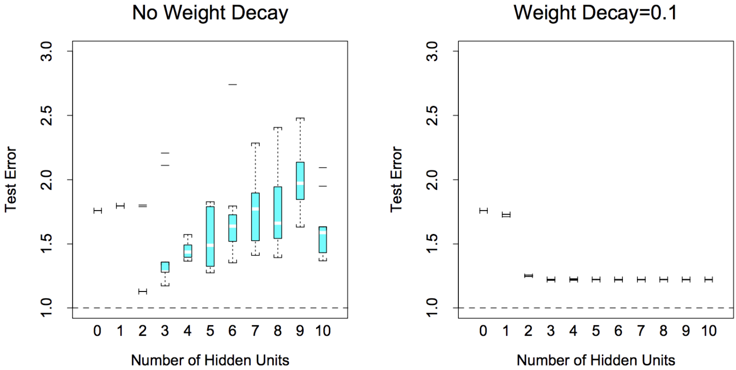
Hastie, T., Tibshirani, R. and Friedman, J. (2009). The elements of statistical learning. New York: springer.

Practical issues with neural networks

- Starting values can heavily influence the performance
  - Different starting weights lead to different solutions due to local minima
  - Typically random values near zero are used
  - Overly large starting values lead to poor performance
- Standardize the inputs before running the algorithm
  - Makes it easier to assign starting values
  - Penalizes all inputs equally
- How many hidden layers / nodes does the neural network have?
  - Using too many is better than too few as long as appropriate regularization is used

Simulated example

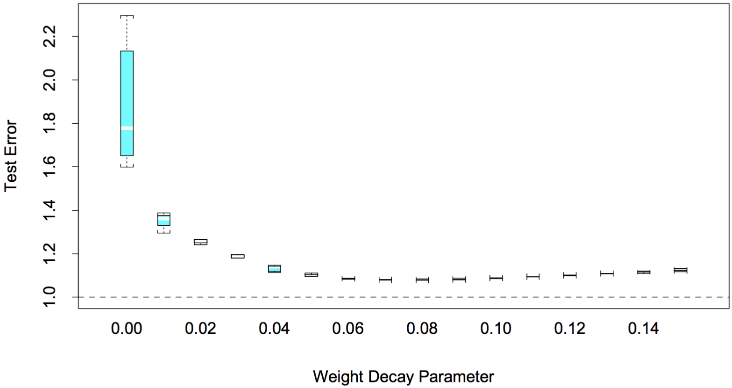
- Here is a simulated example comparing weight decay and no weight decay for a single hidden layer neural network
  - Box plots show the variability across different starting values



Hastie, T., Tibshirani, R. and Friedman, J. (2009). The elements of statistical learning. New York: springer.

Simulated example

- Clearly weight decay has a big impact on the results



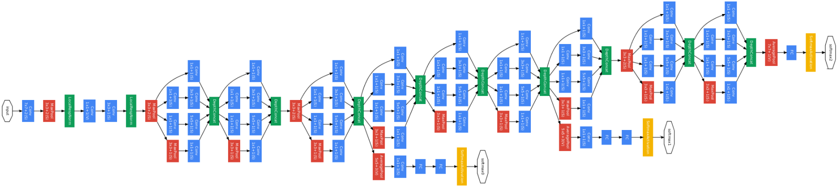
Hastie, T., Tibshirani, R. and Friedman, J. (2009). The elements of statistical learning. New York: springer.

Neural networks

- There are many types of neural networks with varying complexities that are tailored towards certain tasks
- Deep neural networks have many hidden layers, each with potentially many hidden nodes
- Convolutional neural networks are popular for certain uses of neural networks
    - Image classification
    - Pattern recognition

Neural networks

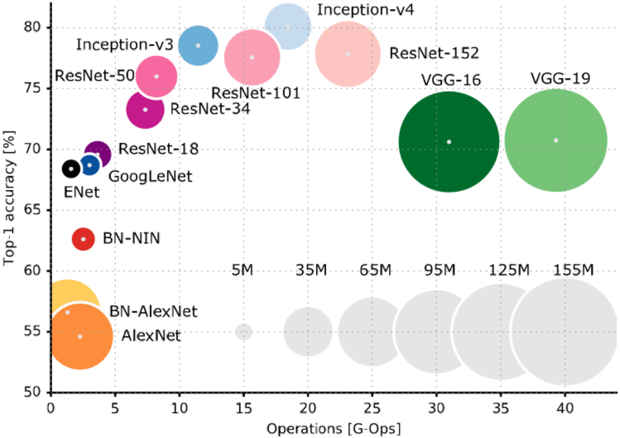
- Many neural networks in practice have a very large set of nodes and hidden layers
- Below is an example of one such network that has 22 layers



Going deeper with convolutions, Szegedy et al. (2015)

Neural networks

- Neural networks with incredibly large numbers of parameters are now being built
  - Some neural networks have over a billion parameters!

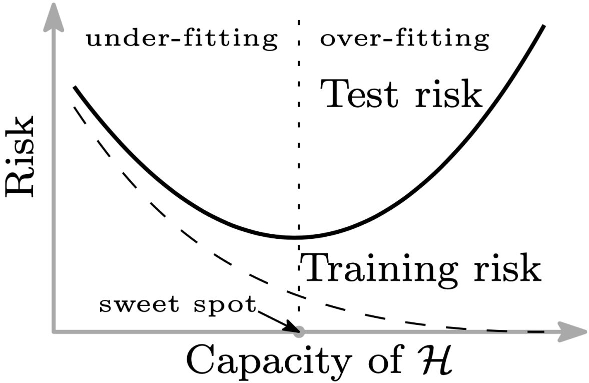


Neural networks

- You might be asking yourself how does this all work
- How can we have so many parameters in our model and avoid overfitting
- Everything so far in our class suggested a bias variance trade-off
  - Too few parameters can lead to bias
  - Too many leads to overfitting and bad generalization to new data
- Interestingly, many neural network models perfectly interpolate the data
  - Having training error zero

Neural networks

- That goes against what we have seen in class, where we usually observed the following

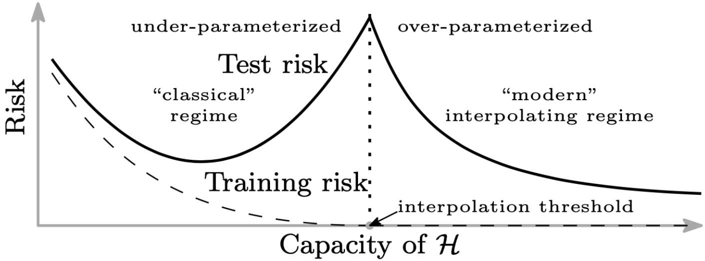


- Training error rates of zero are usually not ideal for a model

Neural networks

- Amazingly, if we go to more complex models that still have training error rates of zero, we can sometimes get improved testing error rates!
  - Called double descent
  - Which descent is lower depends on the specific problem

- The interpolation threshold is the point at which training error becomes zero

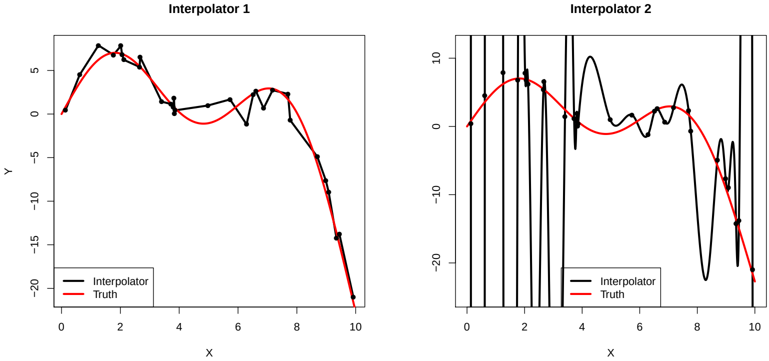


Neural networks

- Whaaaaaaaat?
- How can it be that these models with training error zero that perfectly interpolate the training data will generalize well to new data sets?
- How good or bad an interpolating model is depends on a few things
  - How smooth is the interpolator
  - What is the signal to noise ratio in your model

Interpolators

- Clearly some interpolators are better than others
  - The left one is actually quite good here

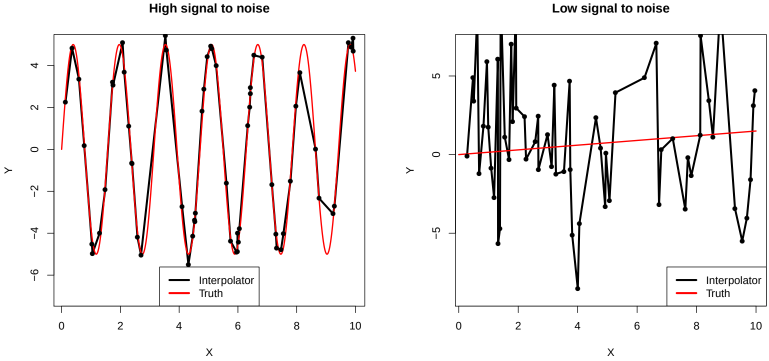


Interpolators

- By considering increasingly complex models, we are considering many possible interpolating functions
- If we choose the one that is the smoothest, then we might end up with a good function that will generalize well
- Whether or not this will outperform simpler models depends on the context
    - Let's highlight now some examples with different signal to noise ratios

Interpolators

- Signal to noise ratio makes a huge impact on the performance of interpolating functions
  - High SNR leads to interpolators that generalize well

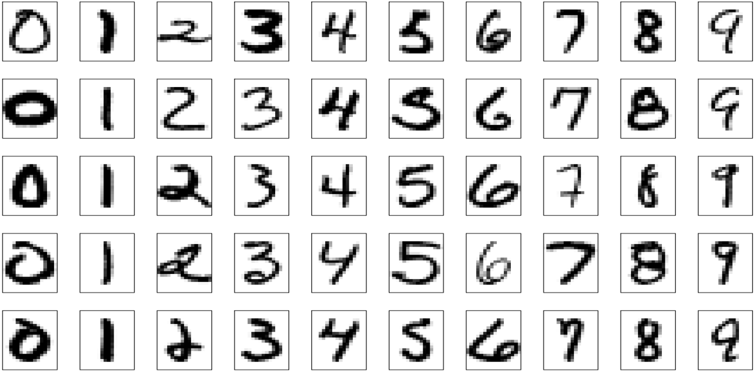


Neural networks

- Neural networks and related complex models tend to work well in situations with a fairly large sample size and high signal to noise ratios
- Can handle extremely complex, nonlinear prediction problems
- Only useful for prediction
    - Neural networks are a black box
    - Don't help us understand the physical processes underlying the data

Neural networks

- Can handle very non-standard types of prediction problems
- A classical data set aims to classify zip code numbers from handwritten numbers



Hastie, T., Tibshirani, R. and Friedman, J. (2009). The elements of statistical learning. New York: springer.

Neural networks

- This image classification is one with a very high signal to noise ratio
  - Clearly we can look at these numbers and classify them correctly almost 100% of the time
- Getting a model to classify this well is a difficult task, but one that neural networks are well suited to
  - The input data are the pixels of an image
  - Correct classification relies on extracting features from these pixels that help in classification
  - Likely these are nonlinear and complex functions of the pixels
- Neural networks have achieved classification rates of over 99% for this problem!

Neural networks

- In summary, neural networks are a very powerful tool in certain situations
- There are many decisions to be made when making and training a neural network that influence performance
- They are not an all encompassing solution for every problem
  - In many cases, simpler models are easier to fit and work better
- We as statisticians, computer scientists, etc. are still trying to better understand these mathematically to gain insight into exactly how they work