SOC 690S: Machine Learning in Causal Inference

Week 3: Machine Learning Advanced

Wenhao Jiang
Department of Sociology, Fall 2025

Duke
UNIVERSITY

Week	Date	Topic	Problem sets	
			Assign	Due
1	Aug 26	Introduction: Motivation and Linear Regression		
2	Sep 2	Foundation: Machine Learning Basics		
3	Sep 9	Foundation: Machine Learning Advanced	1	
4	Sep 16	Foundation: Causal Inference Basics		
5	Sep 23	Foundation: Causal Inference Advanced	2	1
6	Sep 30	Core: PSM and Doubly Robust Estimation		
8	Oct 7	Core: Instrumental Variable Estimation	3	2
7	Oct 14	*Fall break*		
9	Oct 21	**In-class midterm**		
10	Oct 28	Core: Regression Discontinuity Design	4	3
11	Nov 4	Core: Panel Data and Difference-in-Difference		
12	Nov 11	Advanced: Heterogeneous Treatment Effect	5	4
13	Nov 18	Advanced: Unstructured Data Feature Engineering		
14	Nov 25	Advanced: Causal Reasoning in Machine Learning		5
	Dec 2	**Take-home final**		

Bias-Variance Tradeoff

Bias-Variance Tradeoff

- Any function of variables X_i that approximates CEF of Y_i follows the *Bias-Variance Tradeoff* when evaluated using Mean Squared Error (MSE)
- Suppose we have a correct underlying CEF $f(\cdot)$ in the *population* such that

$$\begin{aligned} Y_i &= E[Y_i \mid X_i] + \epsilon_i = f(X_i) + \epsilon_i \\ f(x) &= E[Y_i \mid X_i = x] \\ E[\epsilon_i \mid X_i = x] &= 0 \quad \textit{CEF Decomposition Property} \end{aligned}$$

Bias-Variance Tradeoff

- We use a function $\hat{f}(\cdot)$ fitted using a *sample* to approximate the correct *population* CEF function
- The point-wise expected MSE at $X_i = x$, when sampled from the *population* for a infinite number of times, is defined as

$$MSE(x) = E\left[\left(\hat{f}(x) - f(x)\right)^2 \middle| X_i = x\right] = E\left[\left(\hat{f}(x) - f(x)\right)^2\right]$$

Bias-Variance Tradeoff

- We use a function $\hat{f}(\cdot)$ fitted using a *sample* to approximate the correct *population* CEF function
- The point-wise expected MSE at $X_i = x$, when sampled from the *population* for a infinite number of times, is defined as

$$MSE(x) = E\left[\left(\hat{f}(x) - f(x)\right)^2 \middle| X_i = x\right] = E\left[\left(\hat{f}(x) - f(x)\right)^2\right]$$

- For any random variable Z and any constant a

$$\begin{aligned} E\left[(Z-a)^2\right] &= E\left[Z^2 - 2aZ + a^2\right] \\ &= E[Z^2] - 2aE[Z] + a^2 \\ &= \left(E[Z^2] - (E[Z])^2\right) + \left(E[Z] - a\right)^2 \\ &= \text{Var}(Z) + \left(E[Z] - a\right)^2 \end{aligned}$$

Bias-Variance Tradeoff

- Therefore, $MSE(x)$ can be decomposed as

$$\begin{aligned} MSE(x) &= E\left[\left(\hat{f}(x) - f(x)\right)^2\right] \\ &= \underbrace{V\left(\hat{f}(x)\right)}_{\text{Variance}} + \underbrace{\left(E[\hat{f}(x)] - f(x)\right)^2}_{\text{Bias}^2} \end{aligned}$$

- Intuitively, it means that for any approximation function based on a *sample*, there is always a trade-off between the *variability* of the point-wise estimate (evaluated over repeated *sample* draws) and the *bias* of the point-wise estimate.

Tuning λ in LASSO Regression Addresses the Tradeoff

- LASSO regression minimized the following loss function

$$\hat{\beta}_{LASSO} = \arg\min_{b \in \mathbb{R}^p} \sum_{i}^{n} (Y_i - X_i' b)^2 + \lambda \cdot \sum_{j=1}^{p} |b_j|$$

- At point $X_i = x$ and *penalty level* λ, the prediction is given by

$$\hat{f}_\lambda (X_i = x) = x' \hat{\beta}_{LASSO}$$

- The MSE at $X_i = x$ is then

$$\text{MSE}_\lambda(x) = \underbrace{V\left(\hat{f}_\lambda(x)\right)}_{\uparrow \text{as } \lambda \downarrow} + \underbrace{\left(E\left[\hat{f}_\lambda(x)\right] - f(x)\right)^2}_{\downarrow \text{as } \lambda \downarrow}$$

Tuning λ in LASSO Regression Addresses the Tradeoff

- Remember in K-fold CV, the MSE in a single fold k is defined as[1]

$$MSE_{\lambda}^{k}(X_i) = \frac{1}{m_k} \sum_{i \in B_k} \left(Y_i - \hat{f}^{-k}(X_i; \lambda) \right)^2$$

$$Y_i = f(X_i) + \epsilon_i$$

- We sum up the MSE over all $x \in X_i$ in the left-out *sample* B_k
- Choose λ^* by K-fold CV:

$$\lambda^* = \arg\min_{\lambda} \frac{1}{K} \sum_{k=1}^{K} MSE_{\lambda}^{k}(\lambda)$$

[1]The MSE here has a slightly different meaning; the MSE above is relative to CEF; in the formal definition, it is relative to Y_i. But given the error term is stochastic, one can easily show the two essentially point to the same conclusion.

Advanced Nonlinear Machine Learning

From Linear to Non-Linear Models

- We are interested in predicting outcome Y_i using regressors X_i, which are p-dimensional
- The *best predictor* of Y_i given X_i is the Conditional Expectation Function

$$g(X_i) = E[Y_i \mid X_i]$$

- We have used *linear* prediction, such as OLS and LASSO regression, to approximate $g(X_i)$
- In reality, $g(X_i)$ is hardly linear, but exhibits complex non-linearities
- This motivates more complex models that provide more flexibilities in non-linear approximation

From Linear to Non-Linear Models

- We now introduce two classes of flexible machine learning techniques
- Tree-based Methods and Neural Networks
- Both are *non-linear models* that can capture complex interactions and approximations beyond linear regression
- Tree-based Methods and Neural Networks are commonly used as *nuisance* learners inside *Double Machine Learning* framework that exploit *Neyman Orthogonality*

Tree-based Methods

Tree-based Methods

- Regression trees are based on partitioning the regressor space (the space where X_i takes on values) into a set of *rectangles*
- A simple model is fit within each rectangle
- Since the set of splitting rules used to segment the regressor space can be summarized in a tree, these types of approaches are known as *decision tree* methods

Regression Trees

- The most common approach fits a simple constant model within each rectangle, which corresponds to approximating the unknown function by a *step function*
- Given a partition into J regions denoted $R_1, \ldots, R_J$, the approximation function within each rectangle is given by

$$f(X_i) = \sum_{j=1}^{J} \beta_j \mathbb{1}(X_i \in R_j)$$

  where β_j denotes a constant for each region and $\mathbb{1}(\cdot)$ denotes the indicator function

Regression Trees

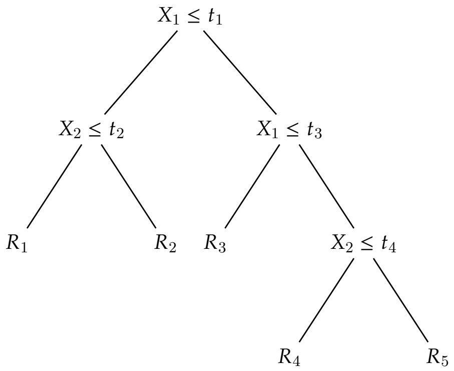
- Suppose we have n observations (X_i, Y_i). The estimated coefficients for a given partition are obtained by minimizing the in-sample MSE

$$\hat{\beta} = \arg\min_{b_1,\dots,b_J} \mathbb{E}\left(Y_i - \sum_{j=1}^{J} \beta_j \mathbb{1}(x \in R_j)\right)^2$$

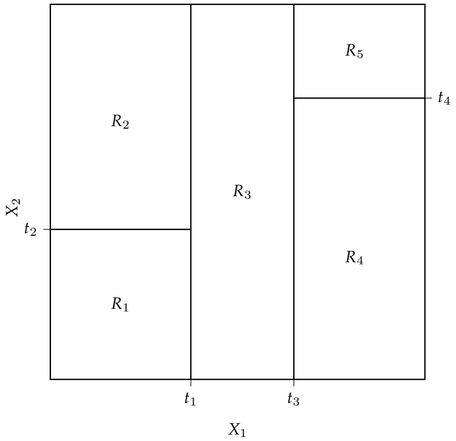
which results in

$$\hat{\beta} = \text{average of } Y_i \text{ where } X_i \in R_j$$

Regression Trees



A partition tree of recursive binary splitting



Output of recursive binary splitting

Regression Trees: Depth 3 Tree in Wage Prediction

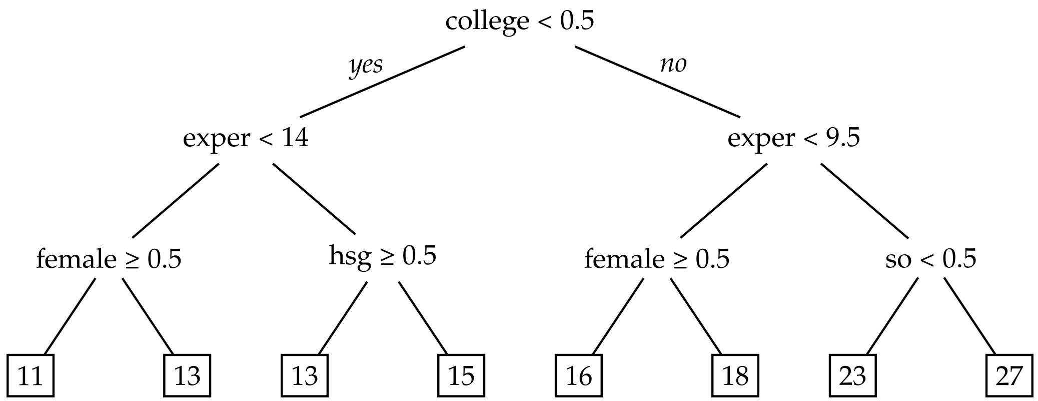


Figure: Depth 3 tree in the wage prediction

Growing a Regression Trees

- The key feature of trees is that the cut points for the partitions are adaptively chosen based on the data
- The splits are *pre-specified* but are purely data dependent
- To make computation tractable, we use recursive binary partitioning or splitting of the regressor space

Growing a Regression Tree

- At each node, consider all predictors $X_1, \dots, X_p$
- For each predictor X_j and threshold s, form a binary split:

$$R_L = \{i : X_{ij} \le s\}, \qquad R_R = \{i : X_{ij} > s\}$$

- Compute the sum of squared errors (SSE) after the split:

$$SSE(s,j) = \sum_{i \in R_L} (y_i - \bar{y}_L)^2 + \sum_{i \in R_R} (y_i - \bar{y}_R)^2$$

  where $\bar{y}_L, \bar{y}_R$ are the mean outcomes in the two child nodes
- Choose (s, j) that minimizes $SSE(s, j)$ (maximizes reduction in MSE)

Growing a Regression Tree

- Recursively execute the procedure
- It is important to note that the splits in level 2 or deeper nodes may use different variables or splitting points
- This features analogously create “interactions” and “nonlinearities”

Growing a Regression Tree

- Recursively execute the procedure
- It is important to note that the splits in level 2 or deeper nodes may use different variables or splitting points
- This features analogously create “interactions” and “nonlinearities”
- We stop growing higher-level nodes when the desired prespecified depth of the tree is reached, or (more commonly) when a prespecified minimal number of observations per region is reached

Growing a Regression Tree

- Why do we care about tree depth

Growing a Regression Tree

- Why do we care about tree depth
- The deeper the tree grows, the smaller the bias, but the noisier (or the higher the variance) becomes, since there are fewer observations per terminal node to estimate the predicted value for this node
- This is the *bias-variance tradeoff* at work
- From a prediction point of view, we can find the "right" depth or the structure of the tree by a validation exercise such as using a single train/test split or cross-validation (CV)
- The process of cutting down the branches of the tree to improve predictive performance is called *tree pruning*

Pruning a Tree

- It is practically not feasible to consider all possible combinations of tree structures and conduct CV
- A better strategy is to grow a “full” tree T_0 (that stops growing when each terminal node has fewer than some minimum number of observations) and *prune* it back in order to obtain a *subtree* T

Pruning a Tree

- *Cost complexity pruning*: consider a tuning penalty parameter λ, there is a *subtree* $T \subset T_0$ that minimizes the in-sample MSE

$$\hat{\beta} = \underset{b_1,\ldots,b_{|T|}}{\arg\min}\, \mathbb{E}\left(Y_i - \sum_{j=1}^{b_{|T|}} \beta_j \mathbb{1}(x \in R_j)\right)^2 + \lambda |T|$$

  where $|T|$ is the number of terminal nodes of the *subtree*
- As in LASSO, the penalty level λ controls the tradeoff between tree complexity and in-sample fit
- Optimal λ and the corresponding tree structured is found by using either a validation set or using CV

Tree-based Methods: Bagging

- Single regression trees can still suffer from *high variance* even with fine-tuning, especially when the sample size is limited
- *Bootstrap aggregation*, or *bagging* is a general-purpose procedure for reducing the variance of a statistical learning method
- Recall that given a set of independent variables $Z_1, \ldots, Z_n$ each with variance σ^2, the variance of the mean $\bar{Z}$ is given by σ^2/n
- We analogously create many parallel estimates of tree-based models, and average the prediction to reduce variance

Tree-based Methods: Bagging

- Specifically, we *bootstrap* the sample for B times, and construct B regression trees
- These trees are grown deep and are *not* pruned
- Hence each individual tree has high variance, but low bias; average these B trees reduces the variance
- Bagging has been demonstrated to give impressive improvements in accuracy by combining together hendreds or even thoughsands of trees into a single procedure

$$\hat{f}_{bag}(X_i) = \frac{1}{B}\sum_{b}\hat{f}^{B}(X_i)$$

Tree-based Methods: Random Forest

- *Random Forests* provide an improvement over bagged trees by *decorrelating* the trees
- As in bagging, we build a number of decision trees on bootstrapped training sample
- But when building these decision trees, each time a split in a tree is considered, *a random sample of m predictors* is chosen as split candidates from the full set of p predictors
- A fresh sample of m predictors is taken at each split, and typically we choose $m \approx \sqrt{p}$

Tree-based Methods: Random Forests

- The intuition is that if there are some relatively highly-correlated predictors, one of which dominates the others in the first split, *bagging* without selection of predictors is likely to produce very similar trees over bootstrapped samples
- Not much new information from other predictors is used with little variance reduction
- *Random Forests* force splits to consider different subsets of predictors to reduce variance

Tree-based Methods: Boosted Trees

- While *bagged trees* and *random forests* reduce variances from low-bias estimates (deep trees) through repetitive sampling, *boosted trees* reduce biases from low-variance estimates (shallow trees) through recursive fitting of residuals
- We estimate a simple prediction rule from a shallow tree, take the *residuals*, and estimate another simple shallow tree for these residuals
- At each step, fitting a model to the residuals from the previous step reduces the approximation error from the previous step
- We keep repeating this process until we reach a stopping criterion
- The sum of these prediction rules at each step gives the overall prediction rule

Tree-based Methods: Boosted Trees

- Initialize the residuals: $R_i := Y_i, i = 1, \ldots, n$
- For $j = 1, \ldots, J$
  - Fit a tree-based prediction rule $\hat{f}_j(X_i)$ to the data $(X_i, R_i)_{i=1}^n$;
  - Update the residuals $R_i := R_i - \lambda \hat{f}_j(X_i)$, where λ is called the *learning rate*
- Output the boosted prediction rule

$$\hat{f}(X_i) := \sum_{j=1}^{J} \lambda \hat{f}_j(X_i)$$

- We need make choices of tree-based prediction rule (depth), the number of learning steps J, and the learning rate (common default $\lambda = 0.1$); these tuning parameters are typically chosen by CV

Tree-based Methods: Boosted Trees

- The number of learning steps J for boosting is important
- Since each step is building a model to predict the unexplained part of the outcome from the previous step, the in-sample prediction errors must weekly increase with each additional step
- If too many iterations are taken, overfitting will likely occur
- While too few iterations may leave significant bias in the final prediction rule
- In practice, the number of iterations is typically chosen by stopping the procedure once there is no marginal improvement to CV-MSE
- A popular implementation widely used in industry is *xgboost*

Neural Networks

Single Layer Neural Network

- A neural network takes an input vector of p variables $X_i = (X_{1i}, X_{2i}, \ldots, X_{pi})$ and builds a nonlinear function $f(X_i)$ to predict the response Y_i
- The p features $X_{1i}, X_{2i}, \ldots, X_{pi}$ make up the units in the *input layer*
- Each of the inputs from the *input layer* feeds into each of the K hidden units (*hidden layer*)
- A single layer neural network has the form

$$f(X_i) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k = \beta_0 + \sum_{k=1}^{K} \beta_k h_k(X_i)$$
$$= \beta_0 + \sum_{k=1}^{K} \beta_k \sigma\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_{ji}\right)$$

Single Layer Neural Network Visualization

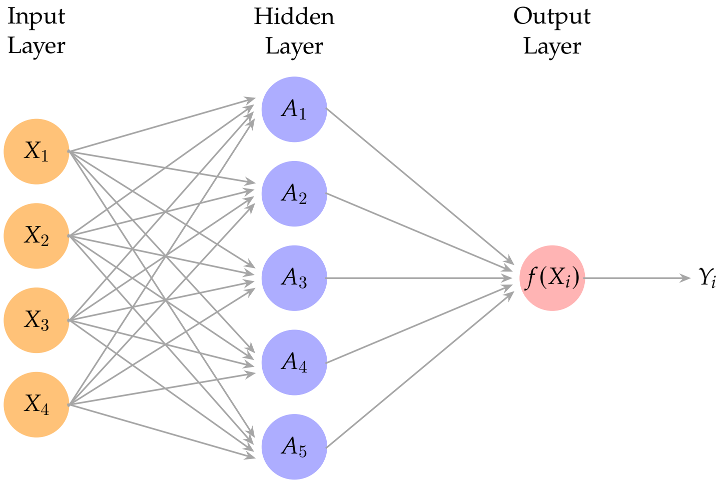


Figure: Neural Network with a single hidden layer

Single Layer Neural Network

- There are two steps in this process
- First the K *activations* $A_k,\ k = 1, \ldots, K$, or *neurons*, in the hidden layer are computed as functions of the input features $X_i, \ldots, X_p$

$$A_k = h_k(X_i) = \sigma\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_{ji}\right)$$

- where $\sigma(\cdot)$ is a nonlinear *activation function* that is specified in advance

Single Layer Neural Network

- There are two steps in this process
- First the K *activations* A_k, $k = 1, \dots, K$, or *neurons*, in the hidden layer are computed as functions of the input features $X_i, \dots, X_p$

$$A_k = h_k(X_i) = \sigma\left(w_{k0} + \sum_{j=1}^{p} w_{kj}X_{ji}\right)$$

- where $\sigma(\cdot)$ is a nonlinear *activation function* that is specified in advance
- These K *activations* or *neurons* from the hidden layer then feed into the output layer, resulting in

$$f(X_i) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k$$

- β_k and w_{kj} are *weights*, and β_0 and w_{k0} are *biases*

Activation Function in Neural Networks

- Popular *activation functions* $\sigma(\cdot)$ include *sigmoid function*

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

- which is the same function used in logistic regression to convert a linear function into probabilities between 0 and 1

Activation Function in Neural Networks

- The preferred choice in modern neural networks is the *ReLU* (*rectified linear unit*) activation function, which takes the form

$$\sigma(z) = (z)_+ = \begin{cases} z & z > 0 \\ 0 & z \leq 0 \end{cases}$$

- A *ReLU* activation can be computed and stored more efficiently than a sigmoid activation; its derivative is simple
- Although it thresholds at 0, the constant term w_{k0} shifts the inflection point
- Both *sigmoid* and *ReLU* provide the property of nonlinearity to the neural network
- Analogous to neurons in the brain: values in the *activation function* need to exceed a threshold to be activated

Nonlinear Activation Provides Nonlinear Prediction

- 2 inputs $X = (X_1, X_2)$ and 1 hidden-layer network

$$f(X) = \beta_0 + \beta_1 h_1(X) + \beta_2 h_2(X)$$
$$h_k(X) = \sigma\left(w_{k0} + w_{k1}X_1 + w_{k2}X_2\right)$$

- Now we choose a nonlinear activation $\sigma(z) = z^2$. Other parameters are

$$\begin{aligned} \beta_0 = 0,\ \beta_1 &= \tfrac{1}{4},\ \ \beta_2 = -\tfrac{1}{4} \\ w_{10} = 0,\ w_{11} &= 1,\ w_{12} = \ \ 1 \\ w_{20} = 0,\ w_{21} &= 1,\ w_{22} = -1 \end{aligned}$$

- The *hidden-layer neurons* are

$$h_1(X) = \left(0 + X_1 + X_2\right)^2 = (X_1 + X_2)^2$$
$$h_2(X) = \left(0 + X_1 - X_2\right)^2 = (X_1 - X_2)^2$$

Nonlinear Activation Provides Nonlinear Prediction

- Plugging the *neurons* in the output layer

$$\begin{aligned} f(X) &= \frac{1}{4}(X_1 + X_2)^2 - \frac{1}{4}(X_1 - X_2)^2 \\ &= \frac{1}{4}\left[\left(X_1^2 + 2X_1X_2 + X_2^2\right) - \left(X_1^2 - 2X_1X_2 + X_2^2\right)\right] \\ &= \frac{1}{4} \cdot 4X_1X_2 = X_1X_2 \end{aligned}$$

Nonlinear Activation Provides Nonlinear Prediction

- A *linear* output layer on top of *nonlinear* hidden units can create *nonlinear functions* (here $X_1 X_2$) even when the base model in X is linear
- With polynomial σ, we recover polynomial features; with *sigmoid* or *ReLU*, we get flexible nonlinear combinations without being limited to fixed polynomial degree
- Without a nonlinear $\sigma(\cdot)$, the network collapses to a linear model in X_i

Nonlinear Activation Provides Nonlinear Prediction

- A *linear* output layer on top of *nonlinear* hidden units can create *nonlinear functions* (here X_1X_2) even when the base model in X is linear
- With polynomial σ, we recover polynomial features; with *sigmoid* or *ReLU*, we get flexible nonlinear combinations without being limited to fixed polynomial degree
- Without a nonlinear $\sigma(\cdot)$, the network collapses to a linear model in X_i
- Hornik, Stinchcombe, and White (1989) and Leshno et al. (1993) proved that multilayer feedforward networks are universal approximators—with a non-polynomial activation function, they can approximate any function

Number of Parameters in a Single Layer Neural Network

- In a single-layer neural network, where there are p inputs and K neurons, the total number of parameters to be estimated is
- From *input* to *hidden* layer $(p+1) \times K$, or $p \times K$ *weights* and K *biases*
- from *hidden* to *output* layer $K+1$, or K *weights* and 1 *bias*
- To minimize the loss function (suppose Y_i is continuous), there are $(p+1) \times K$ *weights* and $K+1$ *biases* to be optimized

$$\underset{W \in \mathbb{R}^{(p+1)\times K};\ \beta \in \mathbb{R}^{K+1}}{\arg\min} \sum_i \left(Y_i - f(X_i)\right)^2$$

Multilayer Neural Network

- Single-layer networks have limited prediction power
- Stacking layers to multilayer neural networks can greatly improve prediction
- Multilayer neural networks famously solved the *MNIST* handwritten-digit problem

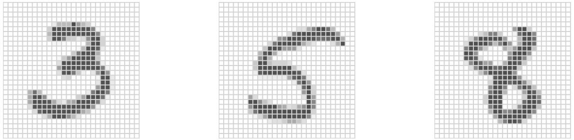
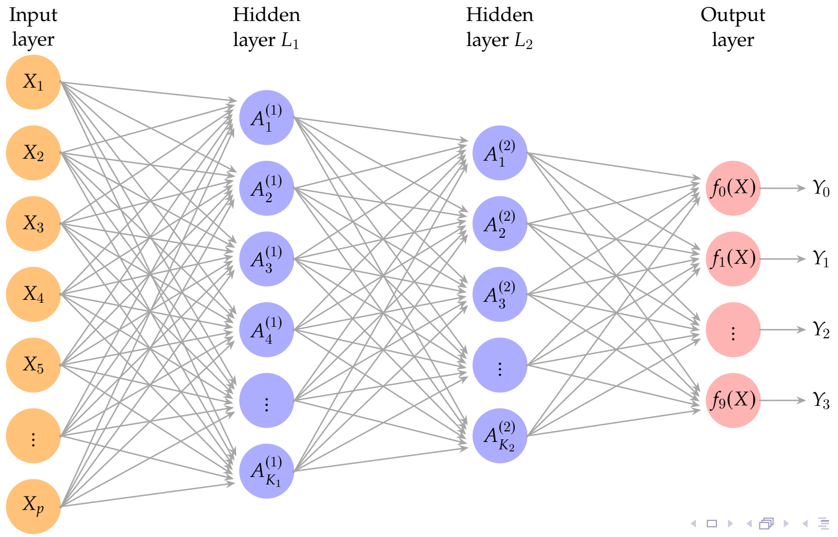


Figure: Examples of handwritten digits from the *MNIST* corpus. Each grayscale image has $28 \times 28 = 784$ pixels, each of which is an 8-bit number (0-255) which represents how dark that pixel is.

Two-Layer Neural Network



Two-Layer Neural Network

- In the *MNIST* problem, *input layer* has $p = 784$ units; *hidden layer* L_1 has $K_1 = 256$ units, *hidden layer* L_2 has $K_2 = 128$ units, and the *output layer* has 10 units corresponding to the probability that the digit is one of the 10 numbers from 0 to 9 based on a *softmax* function
- Each *neuron* $k \in 1, \dots, K_1$ in the first hidden layer L_1 is defined as

$$A_k^{(1)} = h_k^{(1)}(X_i) = \sigma\left(w_{k0}^{(1)} + \sum_{j=1}^{p} w_{kj}^{(1)} X_j\right)$$

- The second hidden layer treats the *neurons* $A_k^{(1)}$ in L_1 as new inputs

$$A_\ell^{(2)} = h_\ell^{(2)}(X_i) = \sigma\left(w_{\ell 0}^{(2)} + \sum_{k=1}^{K_1} w_{\ell k}^{(2)} A_k^{(1)}\right) \quad \text{for } \ell = 1, \dots, K_2$$

Number of Parameters in a Two-Layer Neural Network

- In a two-layer neural network, where there are p inputs, K_1 neurons in the first *hidden layer*, K_2 neurons in the second *hidden layer*, and m units in the *output layer*, the total number of parameters to be estimated is
- From *input* to *first hidden* layer $(p+1) \times K_1$, or $p \times K_1$ *weights* and K_1 *biases*
- From *first hidden* to *second hidden* layer $(K_1+1) \times K_2$, or $K_1 \times K_2$ *weights* and K_2 *biases*
- From *second hidden* to *output* layer, $(K_2+1) \times m$, or $K_2 \times m$ *weights* and m *biases*

Fitting a Neural Network

- We focus on the single layer neural network, which can be extended to multiple layers
- Remember to minimize the loss function (suppose Y_i is continuous), there are $(p+1) \times K$ *weights* and $K+1$ *biases* to be optimized

$$\underset{W \in \mathbb{R}^{(p+1)\times K};\ \beta \in \mathbb{R}^{K+1}}{\arg\min} \sum_i \left(Y_i - f(X_i)\right)^2$$

- This is not straightforward to minimize because
  - With nonlinear *activation* functions, the loss function becomes *non-convex*. Gradient = 0 is necessary but no longer sufficient to find a global minimizer; we may get local minima or saddle points
  - the sheer number of parameters makes the gradient system extremely high-dimensional and unsolvable in closed form in a single calculation

Fitting a Neural Network

- For simplicity, we represent all the parameters in one long vector θ of dimension $(p+1)K + (K+1)$
- Now the loss or *residual* function to be minimized

$$R(\theta) = \frac{1}{2} \sum_{i}^{n} \left(Y_i - f_\theta(X_i)\right)^2$$

- The optimal θ is derived using *gradient descent* that would end up at a good *local* or *global* minimum
- Plus *regularization* to penalize unimportant parameters in case of overfitting

Fitting a Neural Network: Gradient Descent

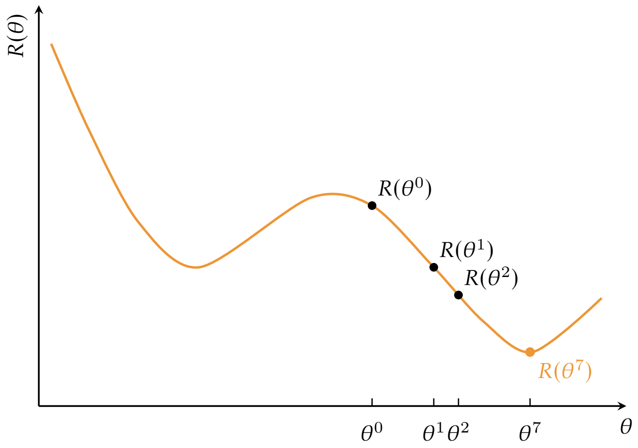


Figure: Illustration of gradient descent for one-dimensional θ. The objective function $R(\theta)$ is not convex, and has two minima, one *local*, the other *global*

Fitting a Neural Network: Gradient Descent

- We start with an initial guess of θ^0 for all the parameters in θ, and set $t = 0$
- Find a vector δ that reflects a small change in θ, such that $\theta^{t+1} = \theta^t + \delta$ *reduces* the loss function; $R(\theta^{t+1}) < R(\theta^t)$
- Stop when the improvement is negligible, *e.g.*,

$$\left|R(\theta^{t+1}) - R(\theta^t)\right| < \epsilon$$

Fitting a Neural Network: Gradient Descent

- We start with an initial guess of θ^0 for all the parameters in θ, and set $t = 0$
- Find a vector δ that reflects a small change in θ, such that $\theta^{t+1} = \theta^t + \delta$ *reduces* the loss function; $R(\theta^{t+1}) < R(\theta^t)$
- Stop when the improvement is negligible, *e.g.*,

$$\left|R(\theta^{t+1}) - R(\theta^t)\right| < \epsilon$$

- δ is derived from the *gradient* of $R(\theta)$ at $\theta = \theta^t$, where the *learning rate* ρ is a small positive constant

$$\begin{aligned} \delta &= -\rho \nabla R(\theta^t) \\ &= -\rho \left.\frac{\partial R(\theta)}{\partial \theta}\right|_{\theta=\theta^t} \end{aligned}$$

Fitting a Neural Network: Gradient Descent

- Calculating the *gradient* is feasible
- To show this, we focus on the loss function $R_i(\theta)$ for one observation i; the total loss is simply the sum over the n observations

$$\begin{aligned} R_i(\theta) &= \frac{1}{2}\big(Y_i - f_\theta(X_i)\big)^2 \\ &= \frac{1}{2}\left(Y_i - \beta_0 - \sum_{k=1}^{K} \beta_k \cdot \sigma\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_{kji}\right)\right)^2 \end{aligned}$$

- For simplicity, we write

$$z_{ik} = w_{k0} + \sum_{j=1}^{p} w_{kj} X_{kji}$$

Fitting a Neural Network: Gradient Descent

- For any β_k, we take the derivative with respect to β_k

$$\begin{aligned}\frac{\partial R_i(\theta)}{\partial \beta_k} &= \frac{\partial R_i(\theta)}{\partial f_\theta(X_i)} \cdot \frac{\partial f_\theta(X_i)}{\partial \beta_k} \\ &= -(Y_i - f_\theta(X_i)) \cdot \sigma(z_{ik})\end{aligned}$$

- For any w_{kj}, we take the derivative

$$\begin{aligned}\frac{\partial R_i(\theta)}{\partial w_{kj}} &= \frac{\partial R_i(\theta)}{\partial f_\theta(X_i)} \cdot \frac{\partial f_\theta(X_i)}{\partial \sigma(z_{ik})} \cdot \frac{\partial \sigma(z_{ik})}{\partial z_{ik}} \cdot \frac{\partial z_{ik}}{\partial w_{kj}} \\ &= -(Y_i - f_\theta(X_i)) \cdot \beta_k \cdot \sigma'(z_{ik}) \cdot X_{kji}\end{aligned}$$

- Notice that both expressions contain the residual $Y_i - f_\theta(X_i)$, meaning that the act of differentiation assigns a fraction of the target residual to each of the parameters via the chain rule—*backpropagation*

Fitting a Neural Network: Stochastic Gradient Descent

- If we execute gradient descent over all n observations at once, the calculation is computationally intensive

$$\theta^{t+1} = \theta^t - \rho \frac{1}{n} \sum_{i=1}^{n} \nabla R_i(\theta)$$

- *Stochastic Gradient Descent* (SGD) instead uses only a subsample of size B_t (called a *mini-batch*) at each update

$$\theta^{t+1} = \theta^t - \rho \frac{1}{|B_t|} \sum_{i \in B_t} \nabla R_i(\theta)$$

- A full pass through all observations (cycling over batches) is called an *epoch*; at the beginning of each epoch, the dataset is shuffled and then split sequentially into non-overlapping *mini-batches*

Fitting a Neural Network: Stochastic Gradient Descent

- *Computational efficiency:* Each update uses only a small batch of data, so updates are much faster than full-batch gradient descent
- *Scalability:* Enables training on massive datasets that cannot fit in memory at once
- *Online learning:* Can incorporate new observations as they arrive, without retraining from scratch
- *Exploration of loss landscape:* The stochasticity (noise) helps the algorithm escape shallow local minima and saddle points
- *Regularization effect:* Noisy updates often prevent overfitting, acting like an implicit form of regularization (approximately quadratic regularization similar to *Ridge*)

Fitting a Neural Network: Dealing with Overfitting

- Just like in high-dimensional data, a massive number of parameters risks overfitting[2]
- A common ℓ_2 Ridge penalty is used to penalize large weights or biases

$$R_i^{Ridge}(\theta) = R_i(\theta) + \lambda \sum_j \theta_j^2$$

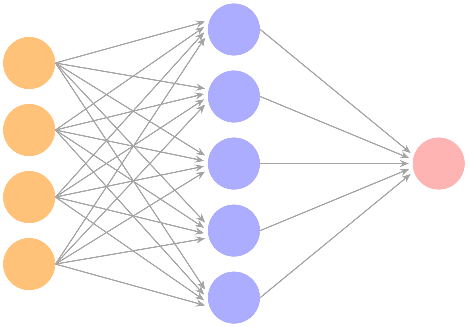
- The SGD for one-observation mini-batch then updates

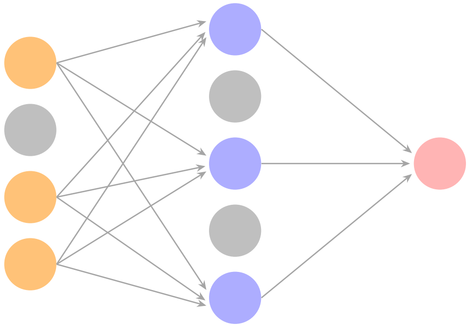
$$\theta^{t+1} \leftarrow \theta^t - \rho\Big(\nabla_{\theta^t} R_i(\theta^t) + 2\lambda\theta^t\Big)$$

[2] $\sum_j \theta_j^2$ means we sum over the square of every scalar entry of the parameter vector θ

Fitting a Neural Network: Dealing with Overfitting

- LASSO-like regularization that filters weights are less common
- Instead, *dropout learning* can prevent nodes from becoming over-specialized


Full Neural Network


Dropout Neural Network

Fitting a Neural Network: Dealing with Overfitting

- In each mini-batch, a fraction ϕ of the units in a layer are randomly dropped (deactivated) during training
- The remaining units are scaled by a factor of $1/(1-\phi)$ so that the expected total input remains unchanged
- In practice, dropout is implemented by randomly setting the activations of the dropped units to zero, while keeping the network architecture intact

Tuning a Neural Network

There are three general classes of hyperparameters one needs to tune for neural networks to perform well

- *Architecture:* number of hidden layers and number of units per layer
- *Regularization:* parameters such as λ (weight penalty) and ϕ (dropout rate), which may be set separately for each layer
- *Optimization:* details of SGD including the learning rate ρ, batch size, and number of epochs

Tuning a Neural Network

There are three general classes of hyperparameters one needs to tune for neural networks to perform well

- *Architecture:* number of hidden layers and number of units per layer
- *Regularization:* parameters such as λ (weight penalty) and ϕ (dropout rate), which may be set separately for each layer
- *Optimization:* details of SGD including the learning rate ρ, batch size, and number of epochs

In practice, for neural networks (especially large ones), CV is often computationally too expensive. Instead, people use a held-out validation set and monitor performance for tuning