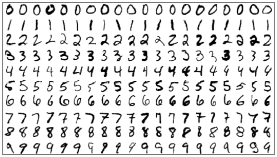
Basics of Machine Learning

COMPSCI 527 — Computer Vision

Outline

A First Classification Problem

- MNIST handwritten digit recognition (60,000 images, labeled, curated)



- 28×28 pixel black-and-white images of individual digits
- What is the *label* $y \in Y = \{0, 1, \ldots, 9\}$ for a given input image x?

Supervised Machine Learning

- Supervised machine learning is the problem of learning a function $\hat{y} = h(\mathbf{x}) \ : \ X \subseteq \mathbb{R}^d \rightarrow Y \subseteq \mathbb{R}$ from sample input/output pairs $(\mathbf{x}, y)$
- "Supervised" means that the samples are provided
- Depending on the problem, *h* may map an image, an image window, or a set of images **x** to
    - A yes or no answer to the question "Is this a [person, car, cat]": $Y = \{\text{yes}, \text{no}\}$ for object detection
    - A category out of a small set: $Y = \{0, 1, \ldots, 9\}$ for digit recognition
    - A category out of a large set: $Y = \{\text{person}, \text{car}, \ldots, \text{tree}\}$ for object recognition
    - A number or small vector of numbers: $Y = \mathbb{R}^5$ for camera motion
    - A whole field (array) of numbers: $Y = \mathbb{R}^{2\times1000\times1000}$ for image motion estimation

Classification and Regression

- Two types of supervised machine learning problems:
- *Classification*: Y is *categorical*, *i.e.*, finite and unstructured

  (For digit recognition, digit-value differences are irrelevant)

  y is then called a *label*
- Examples: Object detection, object recognition, foreground/background segregation
- *Regression*: $Y = \mathbb{R}^e$; y is then called a *value* or *response*
- Distances in Y may be important for learning
- Examples: Camera motion estimation, depth from stereo, image motion estimation, object tracking
- A *target* is either a label or a value

Data Annotation

- The biggest cost in machine learning is *data annotation*
- Manually associate labels to images
- Even harder for regression, where targets are real-valued
- Different annotators may produce different annotations
- May need multiple annotators and take majority votes
- The Amazon Mechanical Turk provides an open market for annotations
- Many companies provide annotation frameworks or services

Why Neural Networks?

- A neural network is a *parametric function*, $\hat{y} = h(\mathbf{x}; \mathbf{w})$
- Parameters in $\mathbf{w} \in \mathbb{R}^m$ are called *weights*
- Neural networks are very *expressive* (large m)
- Can approximate any well-behaved function from a hypercube X in $\mathbb{R}^d$ to a hypercube Y in $\mathbb{R}^e$ within any $\epsilon > 0$
- *Universal approximators*
- However, complexity of approximation grows exponentially with $d = \dim(X)$
- Because m is large, neural networks are *data hungry*: They require large data sets for training

Why Do Neural Networks Work?

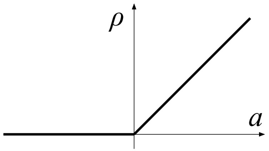
- Theory shows that availability of lots of training data is not a sufficient explanation
- There must be deeper reasons
- Special structure of image space (or audio space)?
- Specialized network architectures?
- Regularization tricks and techniques?
- We don't really know. Stay tuned...
- Be prepared for some hand-waving and empirical statements

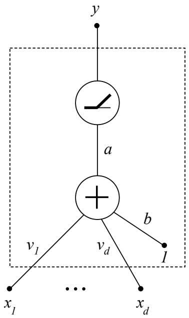
A Generic Deep Neural Network Architecture

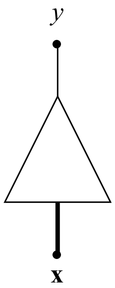
- One basic computational unit, the *neuron*
- Many neurons that receive the same input form a *layer*
- A cascade of layers is a *network*
- A *deep* network has many layers
- Aside on convolutional layers:
    - Layers with a special structure are called *convolutional*
    - We will examine convolutional layers in a later lecture
    - Even convolutional layers fit the generic architecture described next

The Neuron

- $y = \rho(a(\mathbf{x}))$ where $a = \mathbf{v}^T\mathbf{x} + b$
  $\mathbf{x} \in \mathbb{R}^d$, $y \in \mathbb{R}$
- $\mathbf{v}$ are the *gains*, b is the *bias*
- Together, $\mathbf{w} = [\mathbf{v}, b]^T$ are the *weights*
- $\rho(a) = \max(0, a)$ (ReLU, Rectified Linear Unit)

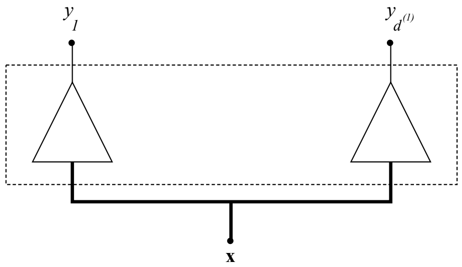


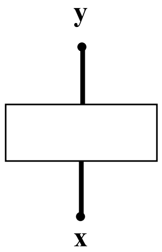




Layers and Networks

- A *layer* is a set of neurons that share the same input





- A *neural network* is a cascade of layers: $\mathbf{y} = \rho(V\mathbf{x} + \mathbf{b})$
- A neural network is *deep* if it has many layers
- *Two* layers can (theoretically) make a universal approximator
- If neurons did not have nonlinearities, any cascade of layers would collapse to a single layer

Using a Regression Network to Classify

- For *regression*, the output of the network is directly the desired quantity (scalar or vector)
- A neural-network *classifier* is designed to have K scalar outputs if there are K labels
- Let $\mathbf{p} = (p_1, \ldots, p_K)$ be the output from the network
- Then, the classification output is $\hat{y} = \arg\max_k p_k$
- Pick the class with the highest score

The Soft-Max Function

- The classification output would be $\hat{y} = \arg\max_k p_k$
- However, a normalized output makes training easier to formulate
- *Soft-max* layer, no trainable parameters:

$$z_k(\mathbf{p}) = \frac{e^{p_k}}{\sum_{j=1}^{K} e^{p_j}}$$

- $z_k(\mathbf{p}) > 0$ and $\sum_{k=1}^{K} z_k(\mathbf{p}) = 1$ for all $\mathbf{p}$
- If $p_i \gg p_j$ for $j \neq i$ then $\sum_{j=1}^{K} e^{p_j} \approx e^{p_i}$
- Therefore, $z_i \approx 1$ and $z_j \approx 0$ for $j \neq i$
- “Brings out the biggest:” *soft-max*
- Since $\arg\max_k z_k = \arg\max_k p_k$, the soft-max layer is often removed after training

Loss

- To know if the learned function h does well on sample $(\mathbf{x}, y)$, we need to measure how far the value $\hat{y}$ it predicts for $\mathbf{x}$ is from the true value y
- The *loss* is a measure of the discrepancy between y and $\hat{y} = h(\mathbf{x})$
- The *loss* function maps pairs $(y, \hat{y})$ to real values: $\ell \; : \; Y \times Y \to \mathbb{R}$
- Simplest loss for classification: the *zero-one loss* or *misclassification loss*
  $$\ell(y, \hat{y}) = I(y \neq \hat{y}) = \begin{cases} 1 & \text{if } y \neq \hat{y} \\ 0 & \text{if } y = \hat{y} \end{cases}$$
- Simplest loss for regression: The *quadratic loss*: $\ell(y, \hat{y}) = (y - \hat{y})^2$
- Different problems call for different measures of loss

Empirical Risk

- Given a *training set* $T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$ with $\mathbf{x}_n \in X$ and $y_n \in Y$, and loss function ℓ, and a neural net architecture $h(\mathbf{x}; \mathbf{w})$ with $\mathbf{w} \in \mathbb{R}^m$, the *empirical risk* or *training error* is the average loss on T: $L_T(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^{N} \ell(y_n, h(\mathbf{x}_n; \mathbf{w}))$
- This is what we minimize in a data fitting problem:

$$\hat{\mathbf{w}} \in \arg\min_{\mathbf{w} \in \mathbb{R}^m} L_T(\mathbf{w})$$

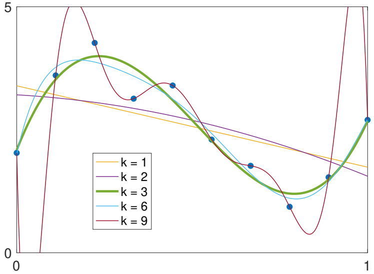
- This is called Empirical Risk Minimization (ERM)

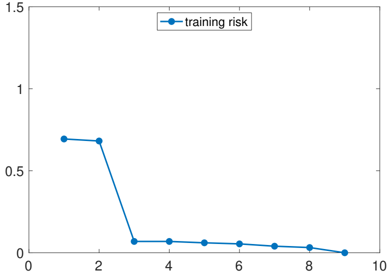
Machine Learning and the Statistical Risk

- ERM: $\hat{\mathbf{w}} \in \arg\min_{\mathbf{w} \in \mathbb{R}^m} L_T(\mathbf{w})$
- **In machine learning, we go much farther: We also want** h **to do well *on previously unseen inputs***
- To relate past and future data, assume that all data comes from the same joint probability distribution $p(\mathbf{x}, y)$
- p is called the *generative data model* or just *model*
- The goal of machine learning is to estimate the *statistical risk* $L_p(\mathbf{w}) = \mathbb{E}_p[\ell(y, h(\mathbf{x}; \mathbf{w}))]$
- p is a good conceptual link between different data sets
- However, p is unknown and cannot be estimated
- Proxy for $L_p(\mathbf{w})$: An empirical risk $L_V(\mathbf{w})$ estimated on a separate *validation set*

SGD, Overfitting, and Underfitting

- $L_T(\mathbf{w})$ is an *average* loss over a *large* training set $\Rightarrow$ SGD!
- Estimate gradient $\nabla L_T(\mathbf{w})$ by $\nabla L_{B_j}(\mathbf{w})$ over mini-batches B_j
- The training risk $L_T(\mathbf{w})$ decreases monotonically as the network learns
- If we keep going, $L_T(\mathbf{w})$ decreases *too much*, often to zero
- Network *overfits*: It learns idiosyncrasies of the data
- Analogous *in concept* to fitting with high-degree polynomial
- But abscissa is different: degree versus training time!

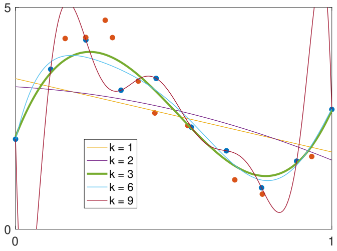


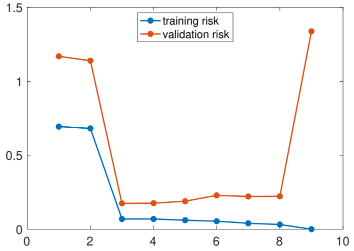


A Separate Validation Set

[Polynomial fitting analogy continued]

- Fit to ("train on") T (blue dots) but evaluate on V (red dots)





- Deep networks: Evaluate on V after each epoch
- Similar plots arise, but with training time instead of degree!
- *Stop training just before the risk on V starts increasing*

Underfitting, Overfitting, and Generalization

- Doing well *on previously unseen data* (V as opposed to T) is the difference between data fitting and machine learning
- Train by reducing the training risk $L_T(\mathbf{w})$ by SGD
- *Stop training just before the validation risk* $L_V(\mathbf{w})$ *starts increasing*
- A predictor with high training risk $L_T(\mathbf{w})$ *underfits*
- A predictor with low training risk $L_T(\mathbf{w})$ and high validation risk $L_V(\mathbf{w})$ *overfits*
- A measure of overfitting is $L_V(\mathbf{w}) - L_T(\mathbf{w})$
- A predictor that neither underfits nor overfits *generalizes well*

Testing

- A machine learning system has been trained, using both T and V, to yield weights $\hat{\mathbf{w}}$ for the network $h(\mathbf{x}; \hat{\mathbf{w}})$
- We cannot report $L_V(\hat{\mathbf{w}})$ as the measure of performance
- The set V is tainted since we used it during training, even if not *for* training
- Performance measures are accepted only on pristine sets, not used in any way for training
- We need to *test* the system on a third set S, the *test set*
- Estimate the statistical risk $L_p(\hat{\mathbf{w}}) = \mathbb{E}_p[\ell(y, h(\mathbf{x}; \hat{\mathbf{w}}))]$ by computing the empirical risk $L_S(\hat{\mathbf{w}}) = \frac{1}{|S|} \sum_{n=1}^{|S|} \ell(y_n, h(\mathbf{x}_n; \hat{\mathbf{w}}))$ on S

Summary of Sets Involved

- A *training set* T to train the predictor
- A *validation set* V to determine when to stop training
- A *test set* S to evaluate the performance of the resulting predictor
- Resampling techniques (“cross-validation”) exist for making the same set play the role of both T and V
- S must still be entirely separate

Training

- Compute $\hat{\mathbf{w}} = \arg\min_{\mathbf{w}\in\mathbb{R}^m} L_T(\mathbf{w})$ by moving along $\nabla L_T(\mathbf{w})$
- $L_T(\mathbf{w})$ is (very) non-convex, so we look for local minima
- Large numbers in $L_T(\mathbf{w})$: m (# of weights) and N (size of T)
- $L_T(\mathbf{w})$ is average of N terms: *Stochastic Gradient Descent*
- Estimate gradient $\nabla L_T(\mathbf{w})$ by $\nabla L_{B_j}(\mathbf{w})$ over mini-batches B_j
- Done by *back-propagation*, which is just the chain rule for differentiation
- The neural network is the chain
- Will see this computation in the next lecture

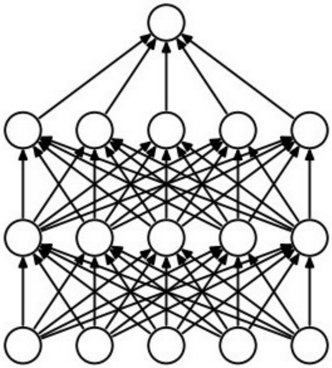
Regularization

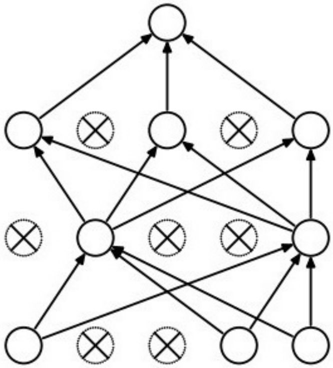
- To further reduce overfitting, most training methods use some type of regularization
- Regularization can be seen as *inductive bias*:

  Bias the training algorithm to find weights in a subset of $\mathbb{R}^m$, rather than in all of it
- Early termination by monitoring the validation risk $L_V(\mathbf{w})$ is regularization, because it does not allow the weights to move arbitrarily far from $\mathbf{w}_0$
- Another method is *weight decay*: add a term $\lambda\|\mathbf{w}\|^2$ to the risk function: Keeps the weights small (Tikhonov)

Dropout

- *Dropout* inspired by ensemble methods:
  Regularize by averaging multiple predictors
- Key difficulty: Too expensive
- Efficient (crude!) approximation:
  - Before processing a new mini-batch, flip a coin with $\mathbb{P}[\text{heads}] = p$ (typically $p = 1/2$) for each neuron
  - Turn off the neurons for which the coin comes up tails
  - Restore all neurons at the end of the mini-batch
  - When training is done, multiply all weights by p
- This is very loosely akin to training a different network for every mini-batch
- Multiplication by p takes the "average" of all networks
- There are flaws in the reasoning, but the method works


(a) Standard Neural Net


(b) After applying dropout.

Data Augmentation

- Data augmentation is not a regularization method, but combats overfitting all the same
- *Make new training data out of thin air*
- Given data sample $(\mathbf{x}, y)$, create perturbed copies $\mathbf{x}_1, \ldots, \mathbf{x}_k$ of $\mathbf{x}$ (these have the same label y)
- Add samples $(\mathbf{x}_1, y), \ldots, (\mathbf{x}_k, y)$ to training set T
- With images this is easy. The $\mathbf{x}_i$s are cropped, rotated, stretched, re-colored, ... versions of $\mathbf{x}$
- One training sample generates k new ones
- T grows by a factor of $k + 1$
- Very effective, used almost universally
- Need to use realistic perturbations