Introduction to Machine Learning

Author: Guanzhou (Jose) Hu 胡冠洲 @ MIT 6.036

Teachers: see the teaching staff list here

Introduction

The **core idea** behind *machine learning* (ML): Instead of "human writing a program to recognize a face" (which is hard), we "write a program that analyzes data to decide how to recognize a face".

High-Level Fundamentals

It does not mean that human engineering becomes less important: instead, human plays a crucial role to *frame* the problem:

- Acquire & Organize data
- Define space of possible solutions
- Characterize objective
- Design & Tune learning algorithms
- Apply algorithm to data (run it)
- Validate the results

Conceptual basis: *problem of induction*, we assume that previously seen data will help us predict the future. Specifically,

- Estimation (statistics)
- Generalization (modeling)

A crucial link is **feature representation**: transform real-world data point x into well-designed features $\phi(x) \in \mathbb{R}_n$.

Different Settings of ML

There are different problem settings of ML:

- *Supervised learning*: we are given list of pairs $D = \{(x_1, y_1), (x_2, y_2), \dots\}$ (i.e., we know the answers of what we have got), and when given a new x', we should be able to predict a y'
    - *Classification*: y_i is an element of a *discrete* set of values
    - *Regression*: y_i is continuous
- *Unsupervised learning*: we are given a data set $D = \{x_1, x_2, \dots\}$, and we are expected to find some patterns or structure inherent in it
    - *Density estimation*: samples are drawn *independent and identically distributed* (i.i.d.) from some distribution Pr, and the goal is to predict $Pr(x')$; sometimes serves as subroutines in other settings
    - *Dimensionality reduction*: the goal is to re-represent samples in lower dimentional space while retaining information
    - *Clustering*: the goal is to find a partitioning (clustering) of the samples that groups "similar" samples
- *Reinforcement learning*: using an *agent* to learn on the way by interacting with the environment w/o prior data (choices affect both the reward & the ability to observe the environment)
- *Sequence learning*: learn a state machine that when given input sequence $x_0, x_1, \dots$, it generates desired output sequence $y_0, y_1, \dots$
- *Semi-supervised learning*: based on supervised, we have an additional set of x_i values with no known y_i, but can still use them to improve the performance

- ...

Supervised Learning Basics

Elements of supervised learning:

- *Hypothesis* (model) $h(\cdot;\theta) \in \mathcal{H}$: $y_i = h(x_i;\theta)$, where θ is the parameters
- *Loss function* $L(g, a)$: how far are we from guess to actual?
- *Training set error* $E_n(h)$: somehow summing over losses on training data, scoring how good is the model h on the training data
  - Example error:

$$E_n = \frac{1}{n}\sum_{i=1}^{n} L(h(x_i;\theta), \hat{y}_i)$$

  - Minimizing traning set error E not necessarily guarantee a small error on testing set!
  - *Corss-validation* can be used when new data is expensive to get
- *Learning algorithm*: taking in data set D and generates (from $\mathcal{H}$) a "best" hypothesis $h(\cdot;\theta)$
  - Be a clever human / Use optimization methods (which should converge)
  - An algorithm can take *hyper-parameters* as the configuration (e.g., # training iterations / error threshold)
  - Parameters are trained during training; Hyper-parameters are adjusted during validation

One thing to remember: evaluating a learning algorithm is different from evaluating a classifier! When we are evaluating whether or not a learning algorithm is good enough - if 100 datapoints can already help us distinguish among those algorithms, it does the work ;)

> Cross validation procedure:
>
> ```
> cross_validate(D, k):
>     divide D into k chunks
>     for i = 1 to k:
>         train h_i on D \ D_i
>         compute "test" error E_i on D_i
>     return averaged E
> ```

ML as Optimization Problem

We can generalize most of the ML problems into *optimization* problems:

- *Object function*: $J(\Theta)$, where Θ captures all the parameters of the model (e.g., θ and θ_0 in perceptron, weight matrices in neural net, ...)
- We want to find $\Theta^* = \arg\min_\Theta J(\Theta)$

The most typical objective function used in ML is:

$$J(\Theta) = \frac{1}{n}\sum_{i=1}^{n} L(h(x_i, \Theta), y_i) + \lambda R(\Theta)$$

- The first term is averaged training error over our training set datapoints
- $R(\Theta)$ is the *regularizer*, which penalizes Θ (using our experience) to relax overfitting
- λ is a hyper-parameter, which governs the degree to which we try to fit on data

Feature Representation

Data points' values in different *dimensions* represent their different **features**.

Quantifying Real World Data

To get data, the first step question would be: how can we quantify and formularize real-world stuff into numeric data that can be used in ML training?

- Real-world is numeric: simple, but pay attention to proper *scaling* (*standardization*: $\tilde{x_i} = \frac{(x_i - \bar{x})}{\sigma_x}$)
- *One-hot* boolean representation for classes
- *Thermometer code* to imply ordering

Factorizing the representations with *domain knowledge* and split separate stages into multiple features helps future learning!

Feature Transformation

How the data are laid out depends on how we represent the features. Some non-separable data under lower dimensions can be easily separated if we take proper transformation of their feature values into higher dimensions. For example, in Perceptron algorithm, not-through-origin linearly separable data in d-dimension can be transformed by appending a 1 to the last dimension and make them through-origin linearly separable in $(d+1)$-dimension.

Transforming feature representation (efficiently & meaningfully) is one of the fundamental parts of ML.

A systematic way of doing this is called *kernel methods*:

- *Polynormial basis*: using k-order basis = include a feature dimension for every possible product of k different dimensions of your original d-dimensional input (with a 1 concatenated)
    - $\Rightarrow$ Resulting # dimensions = # terms in the expansion of $(1 + x_1 + \cdots + x_d)^k$ = # solutions to the equation "$a_0 + a_1 + \cdots + a_d = k$" where a_i are natural numbers = C_{k+d}^{d}
    - e.g., appending a 1 as an extra dimension is equivalent to 1-order polynomial basis
- ...

> DANGER: Transforming into too higher-dimensional space increases the risk of *overfitting*!

Perceptron Algorithm

The classic learning algorithm for linear **classification** (data separatable by a $d-1$ dimensional hyperplane).

Perceptron Training Algorithm

Use (θ, θ_0) to denote a *linear classifier*, that $\theta^T x + \theta_0 = 0$ is a hyperplane, and a point x is classified as *positive* when $\theta^T x + \theta_0 > 0$, vice versa.

- θ is the *norm vector*, perpendicular to the hyperplane, and pointing to the positive side
- $\frac{\theta}{||\theta||}$ is the *unit norm*
- θ_0 is the offset, which allows the hyperplane to not go through the origin

This defines a hypothesis class of linear classifiers $h(x; \theta, \theta_0) = \theta^T x + \theta_0$.

Algorithm pseudocode:

```
PERCEPTRON($\tau, \mathcal{D}_n$)
1  $\theta = \begin{bmatrix} 0 & 0 & \cdots & 0 \end{bmatrix}^T$
2  $\theta_0 = 0$
3  for $t = 1$ to $\tau$
4      for $i = 1$ to $n$
5          if $y^{(i)}\left(\theta^T x^{(i)} + \theta_0\right) \leqslant 0$
6              $\theta = \theta + y^{(i)} x^{(i)}$
7              $\theta_0 = \theta_0 + y^{(i)}$
8  return $\theta, \theta_0$
```

Figure taken from MITx Week2.

Convergence Theorem

A dataset D is said to be **linearly separable** if there is some (θ, θ_0) s.t. all points are classified correctly. Perceptron algorithm converges on datasets with linear separability.

The *margin* of a data point x with respect to a separator (θ, θ_0) is:

$$\text{margin}(x_i) = y_i \cdot \text{signed\_distance} = y_i \cdot \frac{\theta^T x_i + \theta_0}{||\theta||}$$

and the margin of a dataset is then defined as the minimum margin of its members:

$$\text{margin}(D) = \min_i \text{margin}(x_i)$$

The *Perceptron Convergence Theorem* states that,

- If the dataset satisfies:
  - there is θ^* s.t. $\text{margin}(x_i) \geq \gamma > 0$ for all i, and
  - $||x_i|| \leq R$ (all data points contained in a ball of radius R)
- Then perceptron algorithm will make at most $(\frac{R}{\gamma})^2$ mistakes.

Proof can be done by *induction*: denoting $\Theta^{(k)}$ to be the hyperplane after we made k mistakes, $\cos(\theta^*, \theta^{(k)})$ converges to 1.

That said, an extreme initial Θ would still increase # mistake s the algorithm makes during a run.

Linear Logistic Classifier

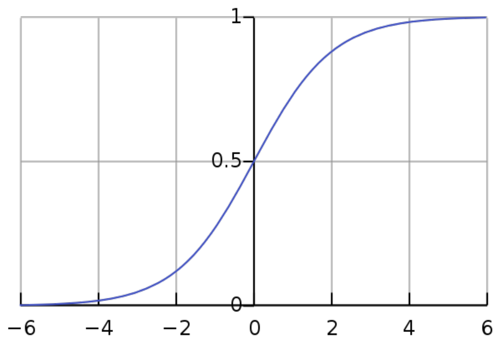
($\equiv$ **Logistic Regression**) Still *linear classification*, but introduces continuity and derivatives, and uses optimization methods to solve.

Sigmoid Function

Loss function used in Perceptron algorithm is a strict *0-1 loss*. However, it is NP-hard to solve a 0-1 optimization problem optimally. Thus, ML people come up with a way to "smooth" this loss: the *Sigmoid function* (*Logistic function*):

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

which "smoothes" out the 0-1 step function, and is differentiable (which gives the algorithm the ability to know "where should I go to make the loss smaller?").

Figure of a standard sigmoid: 

This can give us a new hypothesis class of sigmoid linear separators $h(x; \theta, \theta_0) = \sigma(\theta^T x + \theta_0)$. These separators give a contiguous value in range $(0, 1)$, instead of discrete $\{0, 1\}$ (+, -).

- Setting a threshold value $\sigma(z) \geq T$? reduces it back to a linear separator; $T = 0.5$ gives exactly a 0-1 loss bound
- θ controls how steep the curve is, and θ_0 controls the offset of the central point

Negative Log-Likelihood (NLL) Loss

How can we define a loss function over the outputs of sigmoid function? Think of the sigmoid values as a probability that the datapoint is classified as positive. Then, probability of making all things right is:

$$\Pi_{i=1}^{n} \begin{cases} g_i & \text{if } y_i = 1 \\ 1 - g_i & \text{o.w.} \end{cases}, \text{where } g_i = \sigma(\theta^T x_i + \theta_0),$$

which is totally equivalent to

$$\Pi_{i=1}^{n} g_i^{y_i} (1 - g_i)^{1 - y_i}$$

Because multiplications are hard to take the derivative, we want to convert this formula into a summation. This can be done by taking the log. Then take the negative of it, and it gives the *negative log-likelihood* loss function (also called *log loss / cross entropy*):

$$\sum_{i=1}^{n} L_{nll}(g_i, y_i) = -\sum_{i=1}^{n} (y_i \log(g_i) + (1 - y_i) \log(1 - g_i))$$

Solve by Gradient Descent

Now we can formally defined our objective function:

$$J(\theta, \theta_0; D) = \frac{1}{n} \sum_{i=1}^{n} L_{nll}(\sigma(\theta^T x_i + \theta_0), y_i) + \lambda ||\theta||^2$$

Notice that a regularization term is added here to let the algorithm try a little bit less hard to classify really distant datapoints.

We need a way to solve the optimization problem. The simplest solution is to use *Gradient Descent* (GD) algorithm. Details ~~omitted here~~.

$$\Theta^{(t)} = \Theta^{(t-1)} - \eta \cdot \nabla_{\Theta^{(t-1)}} J,$$

where η is the *learning rate* (*step size*), and iterations stop after error is suffiently small ($<$ threshold ϵ) or after enough # iterations. The gradient for our logistic regression objective function can be simplified as:

$$\begin{cases} \nabla_\theta J & = \frac{1}{n}\sum_{i=1}^n (g_i - y_i)x_i + 2\lambda\theta \\ \frac{\partial J}{\partial \theta_0} & = \frac{1}{n}\sum_{i=1}^n (g_i - y_i) \end{cases}$$

Algorithm pseudocode in 1D:

```
1D-GRADIENT-DESCENT($\Theta_{init}, \eta, f, f', \epsilon$)
1  $\Theta^{(0)} = \Theta_{init}$
2  $t = 0$
3  repeat
4      $t = t + 1$
5      $\Theta^{(t)} = \Theta^{(t-1)} - \eta\, f'(\Theta^{(t-1)})$
6  until $|f(\Theta^{(t)}) - f(\Theta^{(t-1)})| < \epsilon$
7  return $\Theta^{(t)}$
```

Figure taken from MITx Week4.

> (*Batched*) GD is pretty naive in optimization but it works well in ML. Several challenges:
>
> - Falling into local optima (objective function J not convex, η too small)
> - Not converging well (η too large)
> - How to calculate the partial derivatives (gradient)?
>
> Taking the sum at every step is still a little bit expensive. Sometimes, we use *Stochastic* GD (SGD), where in each iteration we randomly pick one dimension of the parameters and only update that dimension. It helps the algorithm to avoid getting stuck in local minima. Also, step size is now a function of # iterations $\eta(t)$ which decreases over time (*adaptive* learning rate), instead of a constant.

Linear Regression

So far, we have covered simple linear classification problems. **Regression** is another class of supervised-learning problems where the outcome is no longer a *discrete* classification but *contiguous* value, i.e., $y_i \in \mathbb{R}$ or $\mathbb{R}^D$.

Ordinary Least Squares (OLS)

In linear regression, the hypothesis is simply $g_i = h(x_i; \theta, \theta_0) = \theta^T x_i + \theta_0$ (we do not take the sign or sigmoid function, because we are not interested in discrete classification anymore). The simplest loss function we measure in linear regression is called the *Squared Error* (SE):

$$L(h_i, y_i) = (h_i - y_i)^2,$$

and it provides the following advantages:

- Penalizes both too-high and too-low predictions
- Easy to take the derivative, and the worse, the bigger the derivative

Thus, the objective function $J(\theta)$ looks like:

$$J(\theta, \theta_0; D) = \frac{1}{n}\sum_{i=1}^{n}(\theta^T x_i + \theta_0 - y_i)^2$$

An *analytical solution* is by taking the derivative with respective to θ and set it to 0, which gives us a *closed-form solution*:

$$\theta^* = (XX^T)^{-1}Xy = (W^TW)^{-1}W^TT$$

Ridge Regression

One problem with the analytical solutioni is that W^TW might not be invertible. To overcome this, we again include a *regularization* term $\lambda||\theta||^2$:

$$J_{\text{ridge}}(\theta, \theta_0; D) = \frac{1}{n}\sum_{i=1}^{n}(\theta^T x_i + \theta_0 - y_i)^2 + \lambda||\theta||^2,$$

and setting its derivative to 0 gives:

$$\theta^*_{\text{ridge}} = (W^TW + n\lambda I)^{-1}W^TT,$$

where the matrix is adjusted by a "ridge" so that it is guaranteed to be invertible. (See MITx Week 5 notes for the detailed procedure of solving the optimal theta.)

Solve by Gradient Descent

Another remaining problem with the analytical solution is that computing the inverse is in the complexity of $O(d^3n)$. To pursue better performance, we apply GD on J_{ridge}:

$$\begin{cases} \nabla_\theta J & = \frac{2}{n}\sum_{i=1}^{n}(\theta^T x_i + \theta_0 - y_i)x_i + 2\lambda\theta \\ \frac{\partial J}{\partial \theta_0} & = \frac{2}{n}\sum_{i=1}^{n}(\theta^T x_i + \theta_0 - y_i) \end{cases}$$

> Since an OLS objective function is *convex*, we are guaranteed to find the optimum. But the complexity is still high.

Stochastic Gradient Descent (SGD)

To further optimize for performance, we introduce *Stochastic Gradient Descent* (SGD). SGD applies when a gradient takes the form of a sum. Instead of computing the whole sum at each step, we randomly sample one piece of the sum and compute the gradient only for it.

$$W = W - \eta \cdot \nabla_W L(h(x_i; W), y_i)$$

Notice that the learning rate η is now a function of t for convergence purpose. Convergence to local optimum is almost ensured if $\sum_{t=1}^{\infty}\eta(t) = \infty$ and $\sum_{t=1}^{\infty}\eta^2(t) = 0$. A simplest choice would be $\eta(t) = \frac{1}{t}$.

> Two parts may contribute to the final test error:
>
> - *Structural Error*: hypothesis class H itself does not contain a good solution
> - *Estimation Error*: not having enough data/not using regularization to specify θ well

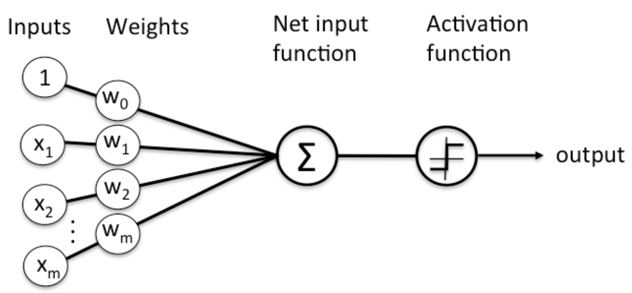
A comprise between whole GD & *stochastic* GD is by using *mini-batches* of size k, called *batched* GD (make sure the data is randomly shuffled):

$$W = W - \eta \cdot \sum_{i=1}^{k}\nabla_W L(h(x_i; W), y_i)$$

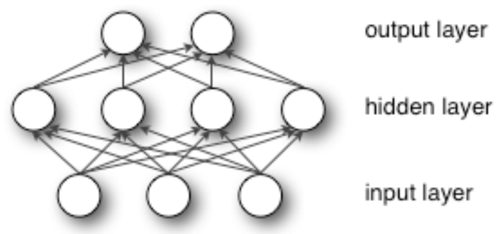
Neural Networks (NN)

Neural Net Basics

老生常谈了，实在不想再整了，盗个图 from this website.

Neuron: 

; Neural net:



Formula:

- Forward feeding:
  - A single neuron - $a = f(z) = f(w^T x + w_0)$
  - A neural net layer - $A^{(L)} = f(Z^{(L)}) = f(W^{(L)T} A^{(L-1)} + W_0^{(L)})$
  - Overall objective - $J(w, w_0) = \sum_i L(NN(x_i; w, w_0), y_i)$
- *Backward propagation*:
  - One layer - $\frac{\partial Loss}{\partial W^{(L)}} = \frac{\partial Loss}{\partial A^{(L)}} \cdot \frac{\partial A^{(L)}}{\partial Z^{(L)}} \cdot \frac{\partial Z^{(L)}}{\partial W^{(L)}} = \frac{\partial Loss}{\partial A^{(L)}} \cdot f(Z^{(L)})' \cdot A^{(L-1)} = \frac{\partial Loss}{\partial Z^{(L+1)}} \cdot W^{(L)T} \cdot f(Z^{(L)})' \cdot A^{(L-1)}$

Weight initialization matters in neural nets because loss - weight function is normally not convex. We can easily fall into *local optima*. It would be better to initialize the weights at random. Making networks bigger is another possible solution.

Whole training process looks like:

```
SGD-NEURAL-NET($\mathcal{D}_n, T, L, (m^1, \dots, m^L), (f^1, \dots, f^L)$)
for $l = 1$ to $L$
    $W^l_{ij} \sim \text{Gaussian}(0, 1/m^l)$
    $W^l_{0j} \sim \text{Gaussian}(0, 1)$
for $t = 1$ to $T$
    $i$ = random sample from $\{1, \dots, n\}$
    $A^0 = x^{(i)}$
    // forward pass to compute the output $A^L$
    for $l = 1$ to $L$
        $Z^l = W^{lT} A^{l-1} + W^l_0$
        $A^l = f^l(Z^l)$
    $loss = \text{Loss}(A^L, y^{(i)})$
    for $l = L$ to 1:
        // error back-propagation
        $\partial loss/\partial A^l$ = if $l < L$ then $\partial loss/\partial Z^{l+1} \cdot \partial Z^{l+1}/\partial A^l$ else $\partial loss/\partial A^L$
        $\partial loss/\partial Z^l = \partial loss/\partial A^l \cdot \partial A^l/\partial Z^l$
        // compute gradient with respect to weights
        $\partial loss/\partial W^l = \partial loss/\partial Z^l \cdot \partial Z^l/\partial W^l$
        $\partial loss/\partial W^l_0 = \partial loss/\partial Z^l \cdot \partial Z^l/\partial W^l_0$
        // stochastic gradient descent update
        $W^l = W^l - \eta(t) \cdot \partial loss/\partial W^l$
        $W^l_0 = W^l_0 - \eta(t) \cdot \partial loss/\partial W^l_0$
```

Figure taken from MITx Week6.

The purpose of *activation function* (non-linear) is that otherwise the net is equivalent to a single linear transformation.

- Output layer activation function should be matched with the loss function we choose for our problem, e.g., using `sigmoid` when our loss is NLL, and using `softmax` when our loss is NLLM
- Choices for other layers: step function, `ReLU`, `sigmoid`, `tanh`, ...

An *epoch* of training is defined as "once through the training data" (or sometimes "once through a sample/mini-batch").

Adaptive Step-size

Adaptive step-size η can better match SGD:

1. *Momentum* averages gradients over time and focus more on recent gradients and less on older gradients:
    - A *running average* of a sequence is defined as $A_t = \gamma_t A_{t-1} + (1-\gamma_t)a_t$:
        - When $\gamma_t = \gamma$ is a constant, it is called a *moving average*: $A_t = \gamma A_{t-1} + (1-\gamma)a_t = \cdots = \sum_{i=1}^{t} \gamma^{t-i}(1-\gamma)a_i$. This average is exponentially more sensitive to recent points
        - When $\gamma_t = \frac{t-1}{t}$, then it is the true average of all data
    - Using the idea of moving average, we define Momentum V as:

$$V_0 = 0; \quad V_t = \gamma V_{t-1} + (1-\gamma)\nabla_W J(W_{t-1})$$

    - Then, we update the weight like:

$$W_t = W_{t-1} - \eta V_t$$

2. *Adagrad* & *Adadelta* introduces the idea that we take big steps when we have low curvature (accelerates the training), and small steps when we have high curvature (easier to fall into the optima):
    - Use a moving average G to record current curvature, G will be high when we are recently going through a curvy region:

    $$G_t = \gamma G_{t-1} + (1-\gamma)(\nabla_W J(W_{t-1}))^2$$

    - Then, we update the weight like:

    $$W_t = W_{t-1} - \frac{\eta}{\sqrt{G_t + \epsilon}} \nabla_W J(W_{t-1})$$

Combining the above two techniques, we get to the *Adam* step-size update strategy:

$$\begin{aligned} g_t &= \nabla_W J(W_{t-1}) \\ m_t &= B_1 m_{t-1} + (1-B_1) g_t \\ v_t &= B_2 m_{t-1} + (1-B_2) g_t^2 \end{aligned}$$

and use the following weight upate rule:

$$W_t = W_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t, \text{ where}$$
$$\hat{m}_t = \frac{m_t}{1-B_1^t}, \hat{v}_t = \frac{v_t}{1-B_2^t}$$

Adam is a practical default used widely nowadays, but under certain circumstances, it may wreck the convergence guarantee.

Regularization against Overfitting

Regularization is another important point to consider when we are using neural nets.

1. *Weight decay*: using the regularizer $\lambda||W||^2$, then weights get updated like $W_t = W_{t-1}(1-\lambda\eta) - \eta \nabla_W J(W_{t-1})$, which potentially helps on shrinking the weight
2. *Early stopping*: as it means, stop early on training data
3. *Perturbing* the data with a little gaussian-distributed perturbation
4. *Dropout*: randomly select a neuron and set its output a_j to 0, thus randomly someone does not affect earlier layers
5. *Batch normalization* (BN):
    - The *gradient explosion/diliminishing* problem: gradient of objective respective to earlier layers' weights is exponential of weights & derivative of activation function
    - Suppose we are doing mini-batches: Interpose a BN box between two layers which processes the output of previous layer as:

    $$X_b^{(i+1)} = \hat{A}_b^{(i)} = \frac{A_b^{(i)} - \mu_b}{\sigma_b}$$

    - Initially address the problem of *covariate shift*:
        - External: dataset's underlying distribution shifts (trained on a specific training set, but applied to a different set with different distribution)
        - Internal: previous layer's weights change, so the next layer is exposed to a different distribution in the next epoch

- Also helps with easing gradient explosion/diliminishing

> Also check *unsupervised learning* (e.g., *autoencoders*) & *semi-supervised learning* (e.g., GNNs).

Convolutional Neural Networks (CNN)

Advanced neural networks take use of our prior knowledge about the input structures / hypothesis space and use that to enrich a standard (*fully connected*) neural net - so that we get better efficiency and performance.

CNNs are a specific kind of neural nets for signal processing scenarios. Suppose inputs are images, they take use of:

- Pixel *space locality*: pixels taken into consideration to find a cat should be near one another
- *Translation invariance*: pattern of pixels that characterizes a cat should be the same no matter where in image the cat occurs

We will focus on images as inputs in the following subsections.

Filters & Channels

A *filter* $=$ a function $f : \text{input\_img} \mapsto \text{output\_img}$. To implement a filter, we normally describe it as a *convolution kernel* and slide this kernel over input pixels - that's how CNN gets its name. Examples:

- 1D left-edge detector: $[-1, +1]$
- 2D edge detector: $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$

> We suppose using *padding* when doing the convolution.

A set of filters used in a CNN filtering layer l is called a *filter bank*:

- number of filters: $m^{(l)}$
- size of each filter: $k \times k \times m^{(l-1)}$ - each filter is a 3D tensor itself!
- stride size: $s^{(l)}$ - we do a convolution every $s^{(l)}$ pixels away

It takes an $n \times n \times m^{(l-1)}$ tensor (i.e., a stack of $m^{(l-1)}$ images) and outputs an $\frac{n}{s^{(l)}} \times \frac{n}{s^{(l)}} \times m^{(l)}$ tensor (i.e., a stack of $m^{(l)}$ shrinked images).

> We normally call a high-dimensional matrix (especially 3D) a *tensor*, e.g., a stack of m same-sized images is a tensor. We sometimes call it m *channels*.

The things to be adjusted in filtering layers are the filter values. Why filters are interesting? Because we only train on the filter values which should be way fewer than using fully-connected weights. Our locality assumption states that the mapping from input image pixels to output image pixels should only take effect locally.

Subsampling

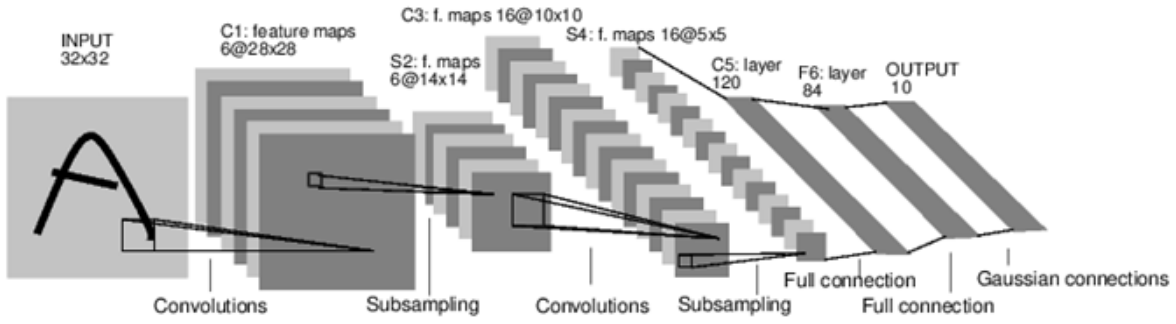
Subsampling layers sample the images and make them smaller. Benefits:

- Extract principal characteristics
- Reduce the computation size

Mostly used in practice are *max / min pooling*. Max pooling by k takes the max value in every $k \times k$ region in original image. This results in a $\frac{n}{k} \times \frac{n}{k}$ image.

Typical CNN Architecture

Typical CNN architecture is as the following (this example is the *LeNet* architecture):



A Full Convolutional Neural Network (LeNet)

The first part is a combination of convolutions (filtering layers) with subsampling layers. After each filtering layer there is normally a ReLU layer. The second part is a normal standard fully-connected neural net. The step from a stack of images to fully-connected layers is called *flattening*.

Back-prop on CNN

Back-propagation on a filtering layer with one filter W of size $k \times 1$ and 1-D input image X of size $n \times 1$ follows:

$$\frac{\partial \text{loss}}{\partial W} = \frac{\partial Z}{\partial W} \cdot \frac{\partial A}{\partial Z} \cdot \frac{\partial \text{loss}}{\partial A},$$

where:

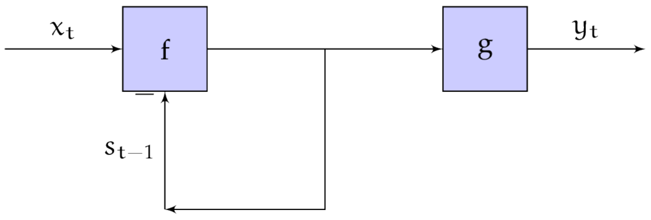
- $\frac{\partial Z}{\partial W}$ is a $k \times n$ matrix where the i, j-th element is $X_{j-\lfloor k/2 \rfloor + i - 1}$
- $\frac{\partial A}{\partial Z}$ is an $n \times n$ diagonal matrix where each diagonal element is a derivative of ReLU
- $\frac{\partial \text{loss}}{\partial A}$ is an $n \times 1$ vector that is the derivative of the loss function

Markov Decision Process (MDP)

Now we go beyond "learning a pure mapping from input to output", and take a look at modeling things over time. These models are called *sequential / recurrent* models.

State Machines

A *state machine* is a 6-tuple (S, X, Y, s_0, f, g):



Two ways of using this structure:

- *Transducer*: takes a sequence of inputs and produces a sequence of outputs

    - Input sequence: $x_1, x_2, \ldots$
    - Output sequence:
        1. $g(f(s_0, x_1)) = g(s_1)$
        2. $g(f(s_1, x_2)) = g(s_2)$
        3. ...
- *Environment*: modeling how a "robot" interacts with an environment
    - Functions now probably not deterministic
    - Example: *Markov Decision Process* (MDP)

Definition of MDP

A *Markov decision process* (MDP) is a special kind of state machine. Instead of a deterministic transition function, the transition function in MDP is a function $T : S \times A \times S \mapsto [0, 1]$, where S is the set of states and A is the set of actions. $T(s, a, s') = P(s_{t+1} = s' \mid s_t = s, A_t = a)$, giving the probability of going to state s' at time $t + 1$, when the old state was s and I took action a.

Key assumption behind this is: $P(s_{t+1} \mid s_0, a_0, s_1, a_1, \ldots, s_t, a_t) = P(s_{t+1} \mid s_t, a_t)$, i.e., the state captures the entire history upto it.

Formally, an MDP is defined as a 4 tuple (S, A, T, R):

- S is the set of all possible states
- A is the set of all possible actions
- T is the transition function defined as above
- $R : S \times A \mapsto \mathbb{R}$ is a *reward* function measuring how worthy is it to take action a on state s

Finite Horizon

A *horizon* represents the number of time steps to go. It maybe finite / infinite. Given a finite horizon and a specific *policy* Π, we define the evaluation of a policy as a *value function* $V^h_\Pi(s)$: the expected sum of rewards given start in state s and execute policy Π for h steps. A recursive way of calculating this:

1. $V^0_\Pi(s) = 0$; $V^1_\Pi(s) = R(s, \Pi(s))$
2. $V^2_\Pi(s) = R(s, \Pi(s)) + \sum_{s'} T(s, \Pi(s), s') \cdot V^1_\Pi(s')$
3. ...
4. $V^h_\Pi(s) = R(s, \Pi(s)) + \sum_{s'} T(s, \Pi(s), s') \cdot V^{h-1}_\Pi(s')$

To efficiently compute $V^h_\Pi(s)$, we may want to use dynamic programming (using memos), filling the table from $V^1_\Pi(s_i)$s to $V^h_\Pi(s_i)$s.

But what about taking different policies at different steps? An optimal policy Π^{h*} when there are h steps in the future is one that satisfies:

$$V^h_{\Pi^{h*}}(s) \geq V^h_{\Pi^h}(s), \text{ for all } s \text{ and possible } \Pi.$$

> There might be multiple optimal policies at a step (they behave equally good).

We then denote the collection of optimal policies for an entire horizon as $\Pi^* = (\Pi^{h*}, \Pi^{(h-1)*}, \ldots, \Pi^{1*})$.

A *Q function* $Q^h(s, a)$ is the expected sum of rewards given start in state s, take action a, and then behave optimally after that (use optimal policies). A similar recursive way of calculating Q:

1. $Q^0(s, a) = 0$; $Q^1(s, a) = R(s, a)$
2. ...

3. $Q^h(s,a) = R(s,a) + \sum_{s'} T(s,a,s') \cdot \max_{a'} Q^{h-1}(s',a') = R(s,a) + \sum_{s'} T(s,a,s') \cdot Q^{h-1}(s', \Pi^{(h-1)*}(s'))$

Equivalently, the optimal policy can be defined as:

$$\Pi^{h*}(s) = \arg\max_a Q^h(s,a)$$

Infinite Horizon

For an infinite horizon, we use a "to-continue probability" γ (the *discount factor*) to model it: every step, we end the game with probability $1-\gamma$.

- Expected length of lifetime $= (1-\gamma)\sum_{n=0}^{\infty}(n+1)\gamma^n = \frac{1}{1-\gamma}$
- A very important feature of this definition is that the expected lifetime does not depend on the time $\Rightarrow$ if we survived today, the expected lifetime is still as long $\Rightarrow$ decision does not change overtime, so we only need one policy and thus one value function

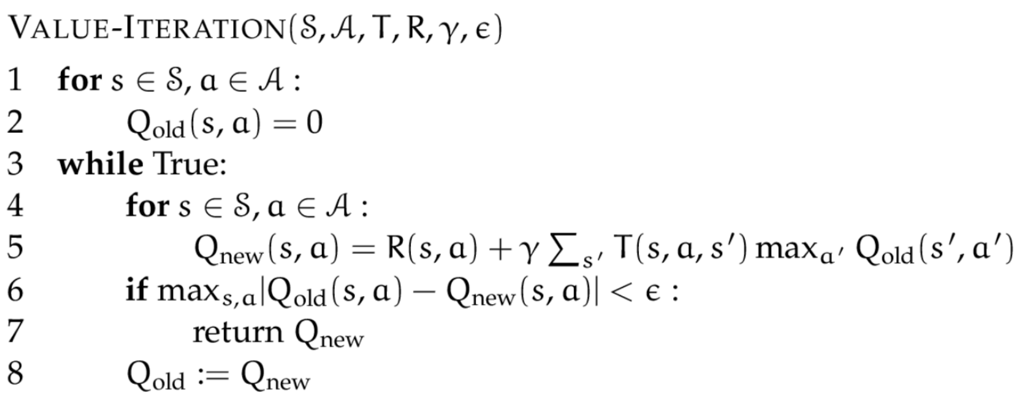
In such settings, the Q function can be defined as:

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') \cdot \max_{a'} Q(s',a')$$

And the optimal policy is:

$$\Pi^*(s) = \arg\max_a Q(s,a)$$

Since now Q depends on the value of Q itself, calculating the values of Q are essentially solving an equation! Thus, we can use a similar idea to *Newton's method* to iteratively solve for these values, called the *value iteration algorithm*:



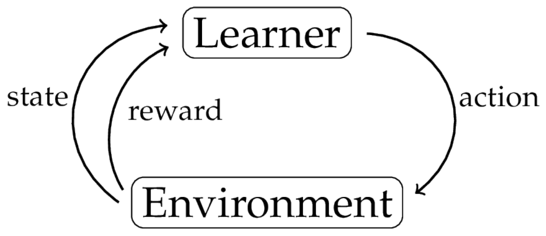
```
VALUE-ITERATION($\mathcal{S}, \mathcal{A}, T, R, \gamma, \epsilon$)
for $s \in \mathcal{S}, a \in \mathcal{A}$:
    $Q_{old}(s,a) = 0$
while True:
    for $s \in \mathcal{S}, a \in \mathcal{A}$:
        $Q_{new}(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q_{old}(s',a')$
    if $\max_{s,a} |Q_{old}(s,a) - Q_{new}(s,a)| < \epsilon$:
        return $Q_{new}$
    $Q_{old} := Q_{new}$
```

Figure taken from MITx Week9.

> Theory guarantees its convergence. We can even run it asynchronously in parallel, and even we use stale intermediate results, it still converges to optimal Q values.

Reinforcement Learning (RL)

Reinforcement learning is a similar setup where an *agent* interacts with the *environment* and tries to learn a good *policy*. The environment (world) here is model as an MDP, but we do not know the exact parameters of that MDP, so we have to learn it while interacting with it.



Bandits & Model-Based Learning

A k-*Armed Bandit* problem is a classic math/statistics problem - a simplified version of this setting. The core tradeoff is *exploration* vs. *exploitation*. Assume a "MDP casino":

- MDP casino with many rooms (states)
- Each room has k bandit machines
- When the agent pull an arm a in room s, it gets a reward $R(s, a)$, and then gets teleported to a room according to some distribution

This is almost the same MDP problem as above. The fundamental difference is that we do not know the rewards and the transition distribution - we have to learn it by exploration.

- *Offline* case: have a free "practice" period when we can ignore the reward values (not getting penalized) and trying to find the optimal policy
- *Online* case: we have the rewards in mind while learning. This is more like real world, but much harder to tackle with

In *model-based learning* with a discrete state space S and action space A, we observe the set of $\{(s_t, a_t, r_t, s_{t+1})\}$ we have experienced, and we estimate the transitions and rewards by:

- $\hat{T}(s, a, s') = \frac{\#(s,a,s')+1}{\#(s,a)+|S|}$ (we are using the *Laplace correction*)
- $\hat{R}(s, a) = \frac{\sum r|s,a}{\#(s,a)}$

Policy Search

In *policy-based learning* (policy search), we pick a parametric form of our policy:

$$a = f(s; \theta),$$

where θ is the parameters which we can train on.

Then, we do gradient descent to minimize the objective function:

$$J(\theta) = -\sum_{t=1}^{T} \gamma^t r_t \mid f(\cdot; \theta)$$

> Policy search works even when the environment is not MDP.

Q-Learning

Q-learning adopts the *value iteration algorithm* on estimated $\hat{R}$ and $\hat{T}$, and a learning rate α:

```
Q-LEARNING($\mathcal{S}, \mathcal{A}, s_0, \gamma, \alpha$)
for $s \in \mathcal{S}, a \in \mathcal{A}$:
    $Q[s, a] = 0$
$s = s_0$ // Or draw an s randomly from $\mathcal{S}$
while True:
    $a = \text{select\_action}(s, Q)$
    $r, s' = \text{execute}(a)$
    $Q[s, a] = (1 - \alpha)Q[s, a] + \alpha(r + \gamma \max_{a'} Q[s', a'])$
    $s = s'$
```

Figure taken from MITx Week10.

There are several different strategies to select an action. A commonly used strategy is called *ϵ-greedy*:

```
def select_action(s, Q):
    with probability 1-eps:
        return argmax_a Q(s, a)      # Greedy.
    else:
        return one uniformly at random
```

What we can further do is that we can use a neural net to approximate the Q function. This is called *function approximation*. The Q update step is then essentially a "gradient descent" training step, with $r + \gamma \max_{a'} Q[s', a']$ as the target value for input $[s, a]$.

> Naturally, we can also do such training in batch, using *experience replay*.

Recurrent Neural Networks (RNN)

Basically, an RNN is a state machine where we use a neural network to implement functions f and g:

- $f(s, x) = f_1(W^{sx}x + W^{ss}s + W_0^s)$
- $g(s) = f_2(W^o s + W_0^o)$

Sequence-to-Sequence Learning

In supervised *sequence-to-sequence learning*, training dataset is a set of pairs of sequences $D = \{(x_i, y_i)\}$, where:

- x_i is a length-n_i sequence of $l \times 1$ vectors
- y_i is a length-n_i sequence of $v \times 1$ vectors

Each pair contains an input sequence and an output sequence of the same length. (Different pairs can have different lengths.) For each pair (which serves as one datapoint), we use the sum of losses of all elements in the sequence as the loss:

$$L_{\text{seq}}(g_i, y_i) = \sum_{t=1}^{n_i} L_{\text{elt}}(g_i[t], y_i[t]),$$

where each single element loss is chosen based on the problem setting. We would like to minimize the following objective function:

$$J(\theta) = \sum_{i=1} L_{\text{seq}}(\text{RNN}(x_i; \theta), y_i)$$

Parameter θ captures all the five weight matrices.

Backpropagation Thorugh Time (BPTT)

Training is done by *Backpropagation Through Time* (BPTT). For an input (a pair of sequences), it "unrolls" the recurrent structure to the length of the sequences.

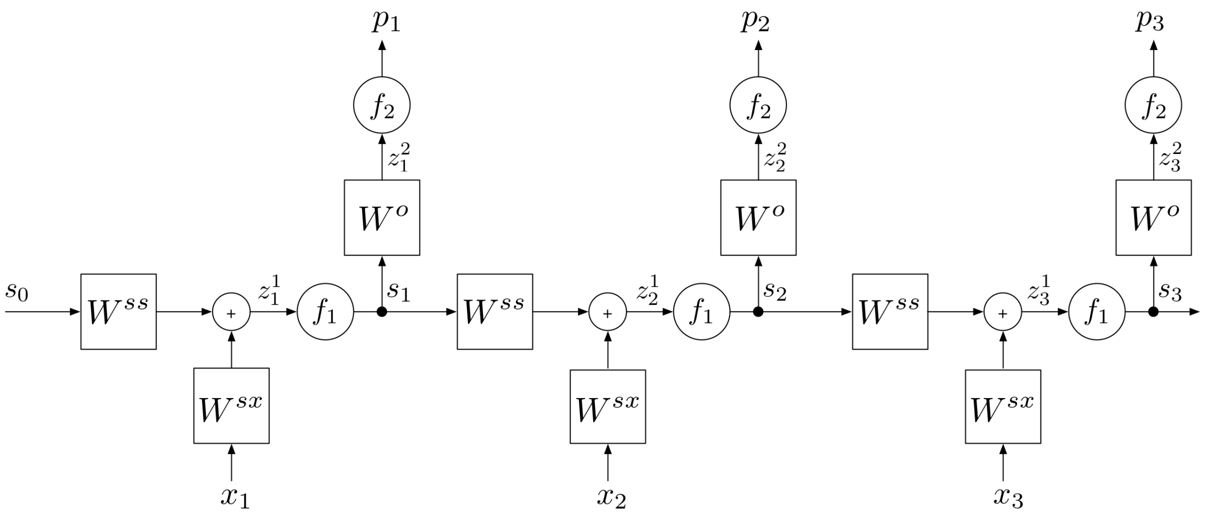


Figure taken from MITx Week11. f_1 in the figure corresponds to f, f_2 in the figure corresponds to g, and p 's are guesses g.

Since the same weight matrices are applied in every time step, the derivatives are a little bit tricky:

$$\begin{aligned}\frac{\partial L_{\text{seq}}}{\partial W} &= \sum_{t=1}^{n} \frac{\partial L_{\text{elt}}(g[t], y[t])}{\partial W} \\ &= \sum_{t=1}^{n}\Big(\sum_{u=1}^{t}\Big(\frac{\partial s[u]}{\partial W} \cdot \frac{\partial L_{\text{elt}}(g[t], y[t])}{\partial s[u]}\Big)\Big) \\ &= \sum_{t=1}^{n}\Big(\frac{\partial s[t]}{\partial W} \cdot \underline{\sum_{u=t}^{n} \frac{\partial L_{\text{elt}}(g[u], y[u])}{\partial s[t]}}\Big) \\ &= \sum_{t=1}^{n}\Big(\frac{\partial s[t]}{\partial W} \cdot \delta^{s[t-1]}\Big)\end{aligned}$$

The underlined part, δ, is defined as:

$$\delta^{s[t]} = \sum_{u=t+1}^{n} \frac{\partial L_{\text{elt}}(g[u], y[u])}{\partial s[t]},$$

which can be computed backward recursively. Base case: $\delta^{s[n]} = 0$. The recursive step looks like:

$$\begin{aligned}\delta^{s[t-1]} &= \frac{\partial s[t]}{\partial s[t-1]} \cdot \Big(\frac{\partial L_{\text{elt}}(g[t], y[t])}{\partial s[t]} + \delta^{s[t]}\Big) \\ &= \frac{\partial Z^1[t]}{\partial s[t-1]} \cdot \frac{\partial s[t]}{\partial Z^1[t]} \cdot \Big(\frac{\partial Z^2[t]}{\partial s[t]} \cdot \frac{\partial A[t]}{\partial Z^2[t]} \cdot \frac{\partial L_{\text{elt}}(g[t], y[t])}{\partial A[t]} + \delta^{s[t]}\Big) \\ &= W^{ss\,T} \cdot f'(Z^1[t]) \cdot (W^{o\,T} \cdot g'(Z^2[t]) \cdot L'(A[t]) + \delta^{s[t]})\end{aligned}$$

In summary, in each epoch, we feed forward, compute the δ's backward, then get the derivatives of loss to weights.

Gating Technique & LSTM

We further introduce the concept of *gates* in RNNs. A gate aims to control, for each component of the state, how much we are taking in the new value and how much we are trying to remember the old value in a time step. More specifically, we compute an extra gate vector:

- $g[t] = \sigma(W^{gx} x[t] + W^{gs} s[t-1])$
- $s[t] = (1 - g[t]) \;.\!*\; s[t-1] + g[t] \;.\!*\; f(W^{sx} x[t] + W^{ss} s[t-1] + W_0^s)$

Long Short-Term Memory (LSTM) is one of the most successful examples of using the gating technique.

Recommender Systems

Recommender systems are an active area of machine learning research and an important example of modern big data technology.

- *Content-based*: Find feature representation $\phi(x)$ of items and predict ratings $\rightarrow$ Supervised regression problem, may suffer from lack of previous ratings data
- *Collaborative filtering*: Use relationships between people and collaboratively adopt others' ratings $\rightarrow$ Maybe GNNs?

  An example simple model would be that we have a sparse dataset Y of $n \times m$ dimension, where we have n users and m items, and we'd like to predict:

  $$X = UV^T,$$

  where:

  - U is $n \times k$, modeling general happiness of each user on each of the k features
  - V is $m \times k$, modeling how much each item contains each of the k features
  - We are modeling X as a rank-k approximation of Y to maintain generality

  Objective function looks like:

  $$J(U,V) = \frac{1}{2} \sum_{(a,i,r) \in D} (U_a V_i + b_{U_a} + b_{V_i} - r)^2 + \frac{\lambda}{2} (\sum_a ||U_a||^2 + \sum_i ||V_i||^2)$$

  We can train by *coordinate descent*, i.e., GD by one coordinate per step, *alternating* between Us and Vs.