4. Linear Neural Networks

Junggwon Shin, shinjunggwon@gmail.com

Summary for Dive Into Deep Learning, https://d2l.ai/chapter_preface/index.html

4.7 Forward Propagation, Backward Propagation, and Computational Graphs

4.8. Numerical Stability and Initialization

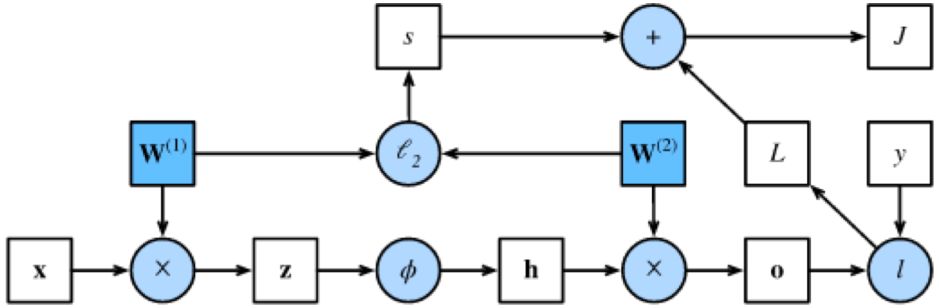
4.9 Environment and Distribution Shift

Training Neural Networks

- When training deep learning models, forward propagation and back propagation are interdependent.
- Forward propagation sequentially calculates and stores intermediate variables within the computational graph
- Backpropagation sequentially calculates and stores the gradients of intermediate variables and parameters within the neural network in the reversed order.
- Training requires significantly more memory than prediction.

Forward Propagation

The calculation and storage of intermediate variables including outputs for a neural network in order from the input layer to the output layer



Computational graph of forward propagation

hidden activation vector of length h

output layer variable with length q

loss function

L2 regularization

objective function

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} \rightarrow \mathbf{h} = \phi(\mathbf{z}). \rightarrow \mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}. \rightarrow L = l(\mathbf{o}, y) \quad s = \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2\right) \rightarrow J = L + s$$

$\mathbf{x} \in \mathbb{R}^d$ input

$\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ 1st weight parameter

ϕ activation function

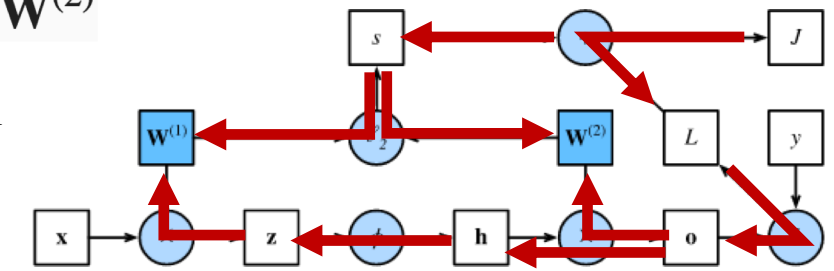
$\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ 2nd weight parameter

$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$ Frobenius norm

Backpropagation

Objective: calculate the gradients of J respect to $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$, $\partial J/\partial \mathbf{W}^{(1)}$ and $\partial J/\partial \mathbf{W}^{(2)}$

Chain rule: $\frac{\partial \mathsf{Z}}{\partial \mathsf{X}} = \text{prod}\left(\frac{\partial \mathsf{Z}}{\partial \mathsf{Y}}, \frac{\partial \mathsf{Y}}{\partial \mathsf{X}}\right)$ prod: function on matrix multiplication for simpler expression



Backpropagation stores only required intermediate variables (partial derivatives) for calculating the gradient of objective function with respect to some parameter

		gradient
	$J = L + s$	$\frac{\partial J}{\partial L} = 1$ and $\frac{\partial J}{\partial s} = 1$.
o: output layer with vector length q	$L = l(\mathbf{o}, y)$	$\frac{\partial J}{\partial \mathbf{o}} = \text{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q$.
$s = \frac{\lambda}{2}\left(\lVert\mathbf{W}^{(1)}\rVert_F^2 + \lVert\mathbf{W}^{(2)}\rVert_F^2\right)$		$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)}$ and $\frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}$.
	$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}$.	$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}}\mathbf{h}^\top + \lambda \mathbf{W}^{(2)}$.
	$\mathbf{h} = \phi(\mathbf{z})$.	$\frac{\partial J}{\partial \mathbf{h}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = \mathbf{W}^{(2)\top}\frac{\partial J}{\partial \mathbf{o}}$.
	$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}$	$\frac{\partial J}{\partial \mathbf{z}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'(\mathbf{z})$. ⊙: elementwise multiplication operator
		$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}}\mathbf{x}^\top + \lambda \mathbf{W}^{(1)}$.

gradient update

$$W_{t+1}^{(2)} = W_t^{(2)} - \alpha \frac{\partial J}{\partial W_t^{(2)}}$$

$$W_{t+1}^{(1)} = W_t^{(1)} - \alpha \frac{\partial J}{\partial W_t^{(1)}}$$

Initialization scheme

- Determine how quickly our optimization algorithm converges and crucial for maintaining numerical stability
- Poor choices of initial values can cause us to encounter exploding or vanishing gradients while training

Vanishing Gradients

- Numerical underflow when multiplying too many matrixes with a wide variety of eigenvalues
- Consider a deep network with L layers, input $\mathbf{x}$ and output $\mathbf{o}$. With each layer l defined by a transformation f parameterized by weights $\mathbf{W}(l)$, whose hidden variable is $\mathbf{h}(l)$ (let $\mathbf{h}(0) = \mathbf{x}$)

$$\mathbf{h}^{(l)} = f_l(\mathbf{h}^{(l-1)}) \text{ and thus } \mathbf{o} = f_L \circ \ldots \circ f_1(\mathbf{x}).$$

$$\partial_{\mathbf{W}^{(l)}} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{(L-1)}} \mathbf{h}^{(L)}}_{\mathbf{M}^{(L)} \stackrel{\text{def}}{=}} \cdot \ldots \cdot \underbrace{\partial_{\mathbf{h}^{(l)}} \mathbf{h}^{(l+1)}}_{\mathbf{M}^{(l+1)} \stackrel{\text{def}}{=}} \underbrace{\partial_{\mathbf{W}^{(l)}} \mathbf{h}^{(l)}}_{\mathbf{v}^{(l)} \stackrel{\text{def}}{=}}.$$

$$\mathbf{o} = \mathbf{h}^{(L)} = \phi\left(\mathbf{W}^{(L-1)}\mathbf{h}^{(L-1)}\right) = \phi\left(\mathbf{W}^{(L-1)}\phi\left(\mathbf{W}^{(L-2)}\mathbf{h}^{(L-2)}\right)\right)$$

$$\frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(L-1)}} = \underbrace{\frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{W}^{(L-1)}}}_{\boldsymbol{v}^{(L)}}, \qquad \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(L-2)}} = \underbrace{\frac{\partial \mathbf{h}^{(L)}}{\partial \mathbf{h}^{(L-1)}}}_{\mathbf{M}^{(L)}} \underbrace{\frac{\partial \mathbf{h}^{(L-1)}}{\partial \mathbf{W}^{(L-1)}}}_{\boldsymbol{v}^{(L-1)}}$$

The gradient is the product of $L-l$ matrices $\mathbf{M}(L) \cdot \ldots \cdot \mathbf{M}(l+1)$ and the gradient vector $\mathbf{v}(l)$

$$\|\mathbf{M}(L)\mathbf{M}(L-1)\| \leq \|\boldsymbol{W}(L)\|\|\boldsymbol{W}(L-1)\| = \boxed{\lambda_{L,1}\lambda_{L-1,1}}$$

- λ_1: the largest eigen(singular)value of a matrix

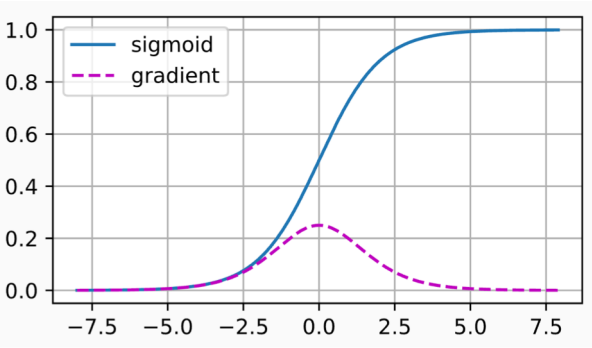
When some values on norm or the largest eigenvalue (spectral radius) of a matrix are extremely small or large
→ gradient vanishing or exploding

Vanishing Gradients (cont.)

- Activation
  - Due to the idea of neurons that fire either fully or not at all gradient, vanishing problem can be caused
- Sigmoid activation function, $1/(1+\exp(-x))$
  - The sigmoid's gradient vanishes both when its inputs are large and when they are small
  - When backpropagating through many layers, the gradients of the overall product may vanish

Exploding Gradients

- For instance, we draw 100 Gaussian random matrices and multiply them with some initial matrix
- When this happens due to the initialization of a deep network, we have no chance to converge gradient descent optimizer



sigmoid function
and its gradient

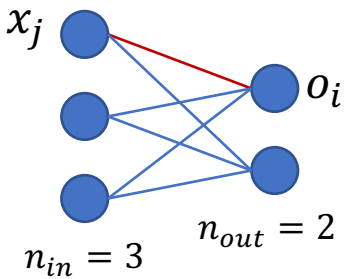
```
M = torch.normal(0, 1, size=(4,4))
print('a single matrix \n',M)
for i in range(100):
    M = torch.mm(M,torch.normal(0, 1, size=(4, 4)))

print('after multiplying 100 matrices\n', M)
```

```
a single matrix
 tensor([[-2.1582e-01,  3.9213e-01, -6.2112e-01, -4.4676e-01],
        [ 9.8879e-01,  7.4085e-01, -6.6903e-01, -4.4329e-01],
        [ 2.5102e-01,  1.7012e-01, -1.0067e+00,  9.1774e-01],
        [ 9.4525e-01, -8.5682e-01, -9.5688e-01,  2.9726e-04]])
after multiplying 100 matrices
 tensor([[-3.4686e+21,  2.7408e+21,  4.7610e+20,  4.6871e+21],
        [ 1.7085e+22, -1.3501e+22, -2.3451e+21, -2.3093e+22],
        [ 2.0956e+22, -1.6560e+22, -2.8765e+21, -2.8324e+22],
        [ 5.4595e+21, -4.3138e+21, -7.4935e+20, -7.3764e+21]])
```

example for illustrations on exploding gradients

Xavier Initialization



- Additional care on initialization of weight parameters to enhance stability of learning
- Scale distribution of an output o_i for some fully-connected layer without nonlinearities.

$$o_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} x_j.$$

- Input x_j, and weights w_{ij} which are all drawn independently from the same distribution
- w_{ij} has zero mean and variance σ^2 and x_j has zero mean and variance γ^2
  - This does not mean that the distribution has to be Gaussian

$\gamma^2 \rightarrow n_{in}\gamma^2\sigma^2$

$n_{out}\tau^2\sigma^2 \leftarrow \tau^2$

variance of input — variance of output

$\gamma^2 = n_{in}\gamma^2\sigma^2$

$n_{in}\sigma^2 = 1$

$$E[o_i] = \sum_{j=1}^{n_{\text{in}}} E[w_{ij} x_j]$$
$$= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}]E[x_j]$$
$$= 0,$$

$$\text{Var}[o_i] = E[o_i^2] - (E[o_i])^2$$
$$= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2 x_j^2] - 0$$
$$= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2]E[x_j^2]$$
$$= n_{\text{in}}\sigma^2\gamma^2.$$

$E[o_i^2] = E\left[\left(\sum_{j=1}^{n_{in}} w_{ij}x_j\right)^2\right] = E\left[\left(w_{i1}x_1 + \cdots + w_{ij}x_j\right)^2\right]$

$= E\left[{w_{i1}}^2{x_1}^2 + \cdots + {w_{ij}}^2{x_{1j}}^2 + 2w_{i1}(x_1 + \cdots + x_j) + \cdots\right]$

E[w] = 0, E[x] = 0

$E\left[{w_{ij}}^2\right] = Var\left[w_{ij}\right] - E\left[w_{ij}\right]^2 = \sigma^2 - 0$

$E\left[{x_j}^2\right] = Var\left[x_j\right] - E\left[x_j\right]^2 = \gamma^2 - 0$

keep the variance fixed

- Objective: keeping **fixed variance between the inputs** with n_{in} values **and the outputs** with n_{out} values to prevent blowing up the gradient's variance, $n_{in}\sigma^2 = 1$
- When considering backpropagation, $n_{out}\sigma^2 = 1$ where n_{out} is the number of outputs of this layer

Xavier Initialization (cont.)

- $n_{in}\sigma^2 = 1, n_{out}\sigma^2 = 1 \rightarrow \frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1 \rightarrow \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}.$

- Typically, Xavier initialization samples weights from a Gaussian distribution with zero mean ~ $N(0, \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}.)$
- Uniform distributions can be applied for weight initialization ~ $U(-a, a)$
  - https://en.wikipedia.org/wiki/Moment-generating_function

The mean (first moment) of the distribution is:

$$E(X) = \frac{1}{2}(a + b).$$

The second moment of the distribution is:

$$E(X^2) = \frac{b^3 - a^3}{3b - 3a}.$$

The variance (second central moment) is:

$$V(X) = \frac{1}{12}(b - a)^2 \longrightarrow \mathrm{V}(X) = \frac{1}{12}(2a)^2 = \frac{1}{3}a^2 = \sigma^2 = \frac{2}{(n_{in} + n_{out})}$$

$$a = \sqrt{\frac{6}{(n_{in} + n_{out})}}$$

- Xavier initialization empirically showed good convergences for sigmoid and tanh activation functions, but poor convergences for ReLU activation
- For Pytorch CNN Layers, Xavier initialization is used as the default

Beyond

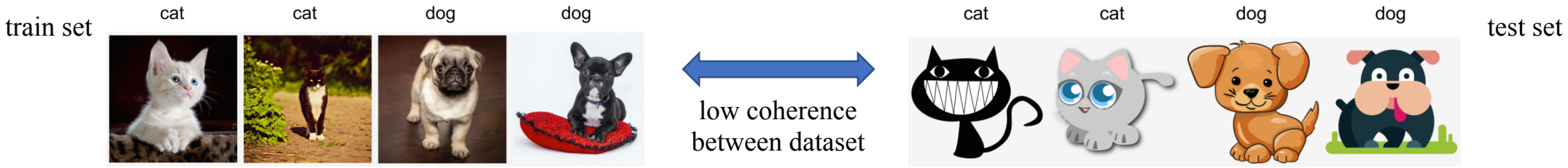
- LeCun initialization: $\sigma = \sqrt{1/n_{in}}$
- He initialization: $\sigma = \sqrt{2/n_{in}}$ (shown effective convergence for ReLU activation)

Distribution Shift

- Change of data distribution after training
  - $P_S(\mathbf{x}, y)$: distribution of train data
  - $P_T(\mathbf{x}, y)$: distribution of test data

Covariate Shift

- Shift in the distribution of the covariates (features, data, $P(x)$), but constant labeling function $P(y|x)$
- Training on a dataset with substantially different characteristics from the test set



Label Shift

- $P(y)$ is changed, but $P(x|y)$ is fixed when label y causes $\mathbf{x}$
- For example, the relative prevalence of data for labels are changing over time

Concept Shift

- Change of labeling function, $P(y|x)$

Correction of distribution shift

- **Empirical risk**: an average loss over the training data to approximate the true risk

$$\underset{f}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} l(f(\mathbf{x}_i), y_i),$$

- **True risk:** expectation of the loss over the entire population of data drawn from their true distribution $p(\mathbf{x},y)$
  - Since we typically cannot obtain the entire population of data, **empirical risk minimization** is used as a strategy for ML

$$E_{p(\mathbf{x},y)}[l(f(\mathbf{x}), y)] = \int \int l(f(\mathbf{x}), y)p(\mathbf{x}, y)\, d\mathbf{x}dy.$$

***Covariate Shift Correction*: $P(x)$**

- $\mathbf{x}$ are drawn from some source distribution $q(\mathbf{x})$ (train set) rather than the target distribution $p(\mathbf{x})$ (real dataset)
- The dependency assumption means that the conditional distribution does not change: $p(y|\mathbf{x}) = q(y|\mathbf{x})$

$$\int \int l(f(\mathbf{x}), y)p(y \mid \mathbf{x})p(\mathbf{x})\, d\mathbf{x}dy = \int \int l(f(\mathbf{x}), y)q(y \mid \mathbf{x})q(\mathbf{x})\frac{p(\mathbf{x})}{q(\mathbf{x})}\, d\mathbf{x}dy.$$

$$p(x, y) = p(y|x)p(x)$$

$$\underset{f}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} \beta_i l(f(\mathbf{x}_i), y_i).$$

$$\beta_i \overset{\text{def}}{=} \frac{p(\mathbf{x}_i)}{q(\mathbf{x}_i)}$$

Weighted empirical risk minimization

Covariate Shift Correction: $P(x)$ *(cont.)*

- For calculate β, we need samples drawn from real time data distribution $p(\mathbf{x})$ and train data distribution $q(\mathbf{x})$
  - Assumption: data example in the target distribution had **nonzero** probability of occurring at training time
- When a data is form $p(\mathrm{x})$, z = 1, otherwise z = -1

$$P(z = 1 \mid \mathbf{x}) = \frac{p(\mathbf{x})}{p(\mathbf{x}) + q(\mathbf{x})} \text{ and hence } \frac{P(z = 1 \mid \mathbf{x})}{P(z = -1 \mid \mathbf{x})} = \frac{p(\mathbf{x})}{q(\mathbf{x})}.$$

- If we use a logistic regression approach, where $P(z = 1 \mid \mathbf{x}) = \frac{1}{1+\exp(-h(\mathbf{x}))}$

$$\beta_i = \frac{1/(1 + \exp(-h(\mathbf{x}_i)))}{\exp(-h(\mathbf{x}_i))/(1 + \exp(-h(\mathbf{x}_i)))} = \exp(h(\mathbf{x}_i)).$$

$$p(z = -1|x) = 1 - p(z = 1|x) = 1 - \frac{1}{1 + \exp(-h(x))} = \frac{\exp(-h(x))}{1 + \exp(-h(x))}$$

- Algorithm
  - Suppose that we have a training set $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ and an unlabeled test set $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_m\}$

1. Generate a binary-classification training set: $\{(\mathbf{x}_1, -1), \ldots, (\mathbf{x}_n, -1), (\mathbf{u}_1, 1), \ldots, (\mathbf{u}_m, 1)\}$.
2. Train a binary classifier using logistic regression to get function h.
3. Weigh training data using $\beta_i = \exp(h(\mathbf{x}_i))$ or better $\beta_i = \min(\exp(h(\mathbf{x}_i)), c)$ for some constant c.
4. Use weights β_i for training on $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ in $\underset{f}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} \beta_i l(f(\mathbf{x}_i), y_i)$.

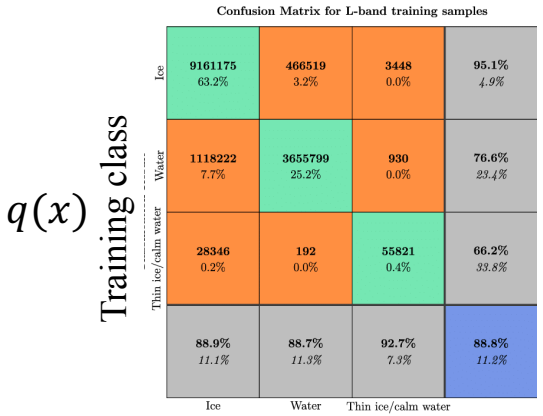
***Label Shift Correction*: $P(y)$**

- $q(y) \neq p(y)$ but the class-conditional distribution stays the same: $q(\mathbf{x} \mid y) = p(\mathbf{x} \mid y)$

$$\int\int l(f(\mathbf{x}), y)p(\mathbf{x} \mid y)p(y)\, d\mathbf{x}dy = \int\int l(f(\mathbf{x}), y)q(\mathbf{x} \mid y)q(y)\frac{p(y)}{q(y)}\, d\mathbf{x}dy. \Rightarrow \beta_i \overset{\text{def}}{=} \frac{p(y_i)}{q(y_i)}.$$

$$p(x, y) = p(x|y)p(y)$$

- Algorithm
  - q(y): just calculate ratio of data for each category
  - p(y): calculate ratio of the predicted results for each category label, $\mu(\hat{y}_i)$ $\Rightarrow$ $\mathbf{C}p(\mathbf{y}) = \mu(\hat{\mathbf{y}}),\ \sum_{j=1}^{k} c_{ij}p(y_j) = \mu(\hat{y}_i)$
    - Since we cannot give the label for test time data unless real-time annotation for the data is possible
  - Construct confusion matrix **C**



1. Generate a binary-classification training set: $\{(\mathbf{x}_1, -1), \ldots, (\mathbf{x}_n, -1), (\mathbf{u}_1, 1), \ldots, (\mathbf{u}_m, 1)\}$.
2. Train a binary classifier using logistic regression to get function h.
3. Weigh training data using $\beta = \mathbf{C}^{-1}$.
4. Use weights β_i for training on $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ in $\underset{f}{\text{minimize}} \frac{1}{n}\sum_{i=1}^{n} \beta_i l(f(\mathbf{x}_i), y_i)$.

***Concept Shift Correction*: $P(y|x)$**

- Concept shift is much harder to fix in a principled manner
  - In computational advertising, new products are launched, but old products become less popular; the distribution over feedback and their popularity changes for each product changes
- In such case, we can use the existing network weights and simply perform a few update steps with the new data rather than training from scratch.

A Taxonomy of Learning Problems

- Batch learning: using to train a model $f(\mathbf{x})$ to score new data $(\mathbf{x}, y)$ drawn from the same distribution
- Online Learning
  - For first observe $\mathbf{x}$, calculate an estimate $f(\mathbf{x})$. After observing y, we receive a reward or loss given our prediction $f(\mathbf{x})$
  - Cycle of online learning

$$\text{model } f_t \longrightarrow \text{data } \mathbf{x}_t \longrightarrow \text{estimate } f_t(\mathbf{x}_t) \longrightarrow \text{observation } y_t \longrightarrow \text{loss } l(y_t, f_t(\mathbf{x}_t)) \longrightarrow \text{model } f_{t+1}$$

  - ex. predicting tomorrow's stock price
- Bandit: searching on finite number of actions for optimal decision making based on theoretical guarantees
- Control: decision making depending on a previous choice. PID (proportional-integral-derivative) controller algorithms are widely used, https://ko.wikipedia.org/wiki/PID_%EC%A0%9C%EC%96%B4%EA%B8%B0
- Reinforcement Learning: learning method considering the Environment (action and state)

Fairness, Accountability, and Transparency in Machine Learning

Accuracy might be a right measure for mission-critical task

Predicting House Prices on Kaggle

https://d2l.ai/chapter_multilayer-perceptrons/kaggle-house-price.html