Exam 2 Review- Foundations of Machine Learning

1. Linear Neural Networks

   Model: $\mathbf{y} = W\mathbf{x} + \mathbf{b}$

   Error function: $E(W, \mathbf{b}) = \sum_{j=1}^{p} \|\mathbf{t}^{(j)} - \mathbf{y}^{(j)}\|^2$

   (a) Widrow-Hoff (Online training)

   Update Rule: $W_{\text{new}} = W_{\text{old}} + \alpha(\mathbf{t} - \mathbf{y})\mathbf{x}^T$ and $\mathbf{b}_{\text{new}} = \mathbf{b}_{\text{old}} + \alpha(\mathbf{t} - \mathbf{y})$

   It is possible to find an optimal learning rate α, but that won't be on the exam.

   (b) Batch training (one-step training)

   If we assume our data is available all at once, we can solve for the weights and biases directly by computing the pseudoinverse. Make the problem linear by appending matrix W with vector $\mathbf{b}$, append a row of 1's in the data matrix X (data points are columns):

   $$\hat{W}\hat{X} = T \quad \Rightarrow \quad \hat{W} = T\hat{X}^{\dagger}$$

2. k-Nearest Neighbors (k-NN)

   Classification: Given a test point x, find the k training points nearest to x, and assign the most common label among them.

   Regression: Predict the average of the values of the k nearest neighbors:

   $$\hat{y}(x) = \frac{1}{k}\sum_{i=1}^{k} y_i$$

   We also discussed how we might use a weighted average of the neighbors. (Do you recall how the weights were set then?)

   Sensitive to: Value of k and the distance function used.

3. In this section, we introduced the important technique of breaking your data into training and testing sets (include a validation set if necessary). The "confusion matrix" was also presented as a good way of displaying the error in a classification problem-it gives more details than just the overall accuracy.

4. We also discussed:

   K-fold cross validation: is used to evaluate the performance of a model more reliably, especially when the dataset is limited in size.

   We use it when we have a limited dataset, or if you're tuning the model parameters and you want a good estimate of your error.

   It helps detect overfitting or underfitting more reliably.

   Recall that k-fold cross-validation gives a better estimate of error by splitting the training data into k parts, training on $k-1$ of them, and then measuring the error on the remaining part—then repeating this k times.

5. Data Clustering

   (a) K-means clustering

   Objective: Minimize within-cluster variance, or distortion error:

$$\min_{\{C_i\},\{\mu_i\}} \sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2$$

Algorithm:

1. Initialize k centroids $\mu_1, \ldots, \mu_k$
2. Assign each x to the nearest μ_i
3. Update each μ_i to be the mean of its cluster
4. Repeat from step 2 until assignments no longer change

The $k-$means algorithm is very fast.

General issues: The mean is sensitive to outliers, the value of k must be given in advance. Works best when data has circular clusters; has a hard time with more complex cluster shapes.

(b) Neural Gas

A "soft" version of k-means where all prototypes (called cluster centers in k-means) are updated according to their ranking:

$$\Delta c_i = \epsilon(t) \cdot e^{-k_i/\lambda(t)} \cdot (x - c_i)$$

where:

- k_i: rank of prototype c_i by distance to x
- $\epsilon(t)$: learning rate at time t
- $\lambda(t)$: neighborhood decay

Additionally, we keep track of which prototypes are connected by keeping an adjacency matrix M and a time matrix T, where

$$M_{ij} = 1 \text{ if centers } c_i \text{ and } c_j \text{ are closest and second closest to } \mathbf{x}$$

$$T_{ij} = \text{ number of iterations since } M_{ij} \text{ was set equal to 1}$$

We assume both M and T are symmetric.

(c) DBSCAN

Density Based Spatial Clustering of Applications with Noise: Most important part of the title is that this clustering is density based.

Parameters: ε (radius), minPts (minimum points to form a dense region)

- **Core point**: at least minPts within ε
- **Border point**: within ε of core, but has $<$ minPts in its own ε-neighborhood
- **Noise point**: not core or border

DBSCAN forms clusters as dense connected regions.

6. Linearization of Functions

Linearizing a function means that we use a linear approximation to a function. In Calc 1, this was a tangent line. In Calc 3, we worked with tangent lines in 3-d (parametric functions) and tangent planes (for $z = f(x, y)$). We want to generalize these results.

(a) $f : \mathbb{R} \to \mathbb{R}$,

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0)$$

(b) $\vec{f} : \mathbb{R} \to \mathbb{R}^n$ (parametric function)

$$\vec{f}(t) \approx \vec{f}(t_0) + \vec{f}'(t_0)(t - t_0)$$

(c) $f : \mathbb{R}^n \to \mathbb{R}$ (like $z = f(x, y)$, for example)

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^T(\mathbf{x} - \mathbf{x}_0)$$

where $\nabla f(x_0)$ is the gradient vector of f at x_0 (as a row vector).
Also for this case, we discussed the Hessian matrix (the matrix of second partial derivatives).

(d) $\vec{f} : \mathbb{R}^n \to \mathbb{R}^m$ (like $\vec{f}(x, y) = \begin{bmatrix} \cos(x) + y \\ x^2 \\ 2x + 3y \end{bmatrix}$, for example)

$$\vec{f}(\mathbf{x}) \approx \vec{f}(\mathbf{x}_0) + Jf(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)$$

where $Jf(x_0)$ is the $m \times n$ matrix of first partial derivatives (the rows are the gradients of each coordinate function).

7. Root Finding Algorithms

(a) Bisection method
Given f continuous on $[a, b]$ with $f(a)f(b) < 0$, the root is approximated by repeatedly halving the interval.

$$x_{n+1} = \frac{a_n + b_n}{2}$$

If $f(x_{n+1})f(a_n) < 0$, set $b_{n+1} = x_{n+1}$, else $a_{n+1} = x_{n+1}$. Converges linearly.

(b) Newton's Method
Given a differentiable function f, iterate:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Derivation: From the linearization $f(x) \approx f(x_n) + f'(x_n)(x - x_n)$, solve $f(x) = 0$:

$$0 \approx f(x_n) + f'(x_n)(x - x_n) \Rightarrow x = x_n - \frac{f(x_n)}{f'(x_n)}$$

(c) Multivariate Newton's Method
Let $G : \mathbb{R}^n \to \mathbb{R}^n$. Then:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - JG(\mathbf{x}_n)^{-1}G(\mathbf{x}_n)$$

Derivation: Linearize G using the Jacobian matrix JG:

$$G(\mathbf{x}) \approx G(\mathbf{x}_n) + JG(\mathbf{x}_n)(\mathbf{x} - \mathbf{x}_n)$$

Set $G(\mathbf{x}) = 0$ and solve:

$$0 = G(\mathbf{x}_n) + JG(\mathbf{x}_n)(\mathbf{x} - \mathbf{x}_n) \Rightarrow \mathbf{x} = \mathbf{x}_n - JG(\mathbf{x}_n)^{-1}G(\mathbf{x}_n)$$

Computationally, if $\vec{\delta} = JG(\mathbf{x}_n)^{-1}G(\mathbf{x}_n)$, it is better to solve the system of equations given by: $JG(\mathbf{x}_n)\vec{\delta} = G(\mathbf{x}_n)$

8. Optimization

(a) Gradient Descent

Given a differentiable function $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$, and an initial guess, $\mathbf{x}_0$, we iterate

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \eta \nabla f(\mathbf{x}_n)^T$$

where the gradient itself is assumed to be a row vector, and η is the learning rate.

In machine learning, we'll be minimizing the error function which will depend on the model parameters.

Derivation:

To show that the direction of fastest increase from a given point $\mathbf{x} = \mathbf{a}$ is in the direction of the gradient, $\nabla f(\mathbf{a})$, recall that

$$\mathbf{x} \cdot \mathbf{y} = \|\mathbf{x}\| \|\mathbf{y}\| \cos(\theta)$$

Secondly, recall that the rate of change of f in the direction of unit vector $\mathbf{u}$ (from point $\mathbf{a}$) is the directional derivative:

$$D_u f(\mathbf{a}) = \nabla f(\mathbf{a}) \cdot \mathbf{u}$$

Therefore,

$$D_u f(\mathbf{a}) = \nabla f(\mathbf{a}) \cdot \mathbf{u} = \|\nabla f(\mathbf{a})\| \cos(\theta)$$

The expression on the right is maximized when $\cos(\theta) = 1$, or when $\theta = 0$. That means the $\mathbf{u}$ points in the direction of the gradient.

On the step size or learning rate

Although we can find the optimum value of the learning rate by optimizing our function on a line through $\mathbf{a}$ in the direction of $\nabla f(\mathbf{a})$, I won't ask you to do that on the exam. You should be able to write the equation of that line, if I give you a function and a point (like in the exercises).

(b) Stochastic Gradient Descent (SGD)

The method is used to minimize an error function that typically depends on a lot of data points. In this case, to compute the full gradient of the error would mean that we would need to use all the (training) data.

In Stochastic Gradient Descent, we use a sample (or mini-batch) from the dataset to compute an **approximate** gradient:

$$\mathbf{z}_{n+1} = \mathbf{z}_n - \eta \nabla E_i(\mathbf{z}_n)$$

where E_i is the error function for a sample i, and the vector $\mathbf{z}$ contains the training parameters (in linear regression, for example, $\mathbf{z}$ would hold the slope and intercept). Introduces noise, useful for large datasets and escaping local minima.

What to study

I may ask you to take one step through the algorithm by hand using a simple function, like the first homework question on page 116 of the notes.