ECE 479 ICC: IoT and Cognitive Computing
Spring 2024, Homework 2

Yanghonghui Chen /yc47

Question 1: K-Means Clustering [20 pts]

1. **Clustering concepts [5 pts]**

   (a) Clustering does not require prior labeling, thus it is a kind of unsupervised learning. Please answer True or False. **[1 pts]**

   True

   (b) The goal of all clustering algorithms is to minimize the total within-cluster SSEs (sum of squared error). Please answer True or False and explain your answer. **[2 pts]**

   False.
   Explanation: While minimizing the total within-cluster sum of squared errors (SSE) is a common objective in many clustering algorithms like K-means, it's not the goal for all clustering algorithms. Different clustering algorithms may have different optimization objectives. For example, hierarchical clustering aims to minimize or maximize linkage distances, while density-based clustering methods aim to identify regions of high density. Therefore, the statement is false because the objective can vary depending on the algorithm used.

   (c) It is impossible to define similarity in terms of clustering for categorical data. Please answer True or False and explain your answer. **[2 pts]**

   False.
   Explanation: It is possible to define similarity for categorical data in clustering, although the method differs from similarity measurement in numerical data. For categorical data, similarity can be defined based on metrics such as the Jaccard similarity coefficient, which measures the similarity between two sets by comparing their intersection to their union. Other metrics like

the Hamming distance or the cosine similarity can also be used depending on the nature of the categorical data. Therefore, the statement is false because similarity can indeed be defined for categorical data in clustering, albeit using different metrics than for numerical data.

2. **Clustering applications [6 pts]**
   Please list at least **three real applications** of the clustering algorithm. Each of them should include three to four sentences:

   (a) a brief description of the application

   (b) a reference to support your statement (URL link or title of the articles, etc.)

   (1) Image Segmentation in Medical Imaging:

   - Image segmentation is the process of partitioning an image into multiple segments or regions to simplify its representation or aid in the analysis of medical images. Clustering algorithms, particularly fuzzy C-means clustering or spectral clustering, are used in medical imaging to segment anatomical structures or identify abnormalities in images such as MRI scans, CT scans, or X-rays. Accurate segmentation enables physicians to visualize and analyze specific regions of interest, leading to better diagnosis and treatment planning.
   - Reference:"Brain tumor segmentation based on deep learning and an attention mechanism using MRI multi-modalities brain images"

   (2) Customer Segmentation in Marketing:

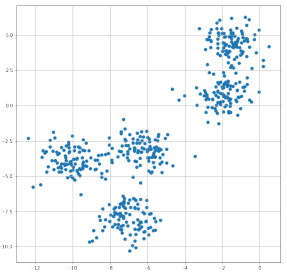
   - Customer segmentation involves dividing a customer base into groups that share similar characteristics or behavior. Clustering algorithms such as K-means or hierarchical clustering can be used to segment customers based on variables like demographics, purchase history, or browsing behavior. This segmentation helps marketers tailor their marketing strategies and product offerings to specific customer segments, leading to more personalized and effective marketing campaigns.
   - Reference:"Customer Segmentation using K-means Clustering"

   (3)Document Clustering in Information Retrieval:

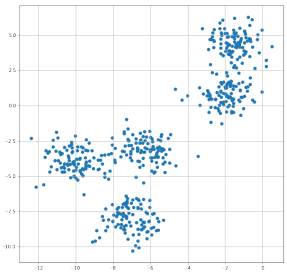
   - Document clustering involves organizing a large collection of documents into clusters based on their content similarity. Clustering algorithms like hierarchical clustering or latent Dirichlet allocation (LDA) are commonly used in information retrieval systems to group related documents together. This facilitates efficient document organization, browsing, and retrieval, enabling users to find relevant information quickly and effectively.

- Reference:"Latent Dirichlet Allocation"

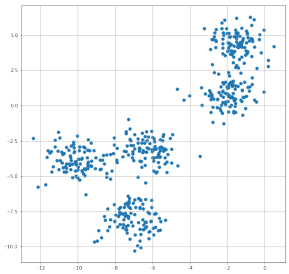
3. **Visualizing a K-means problem [4 pts]**
   This part of the question aims to help you build an intuitive sense of how the "K-means algorithm" works. The following data points are randomly generated with a pre-set number of centroids.
   In the following diagrams, try to circle out clusters according to the given K in each of them and then comment on the results. Give your best guess on what should be the optimal K, and explain the reason using the "SSE vs K" diagram. (Discussed in the Lecture and please refer to Figure 2. For example: if "K=1", you should circle all data points to be the same cluster since there is only 1 cluster.)
   Answer:
   Based on the "SSE vs K" diagram, there are sharp drops of SSE for K=2,3,4,5, and after K=5, the change of SSE becomes flat which indicates that there are no more distinguished classes.



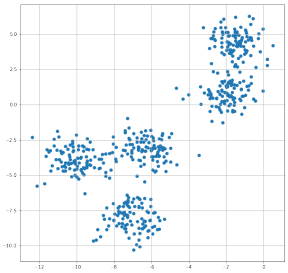
(a) $K = 2$



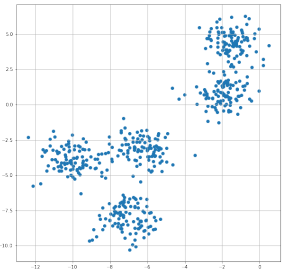
(b) $K = 3$



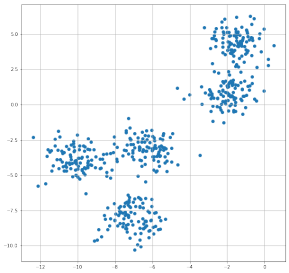
(c) $K = 4$



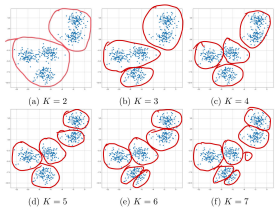
(d) $K = 5$



(e) $K = 6$



(f) $K = 7$



(g) *Answer*

4. **Calculating and minimizing SSE [5 pts]** In this question, calculate the

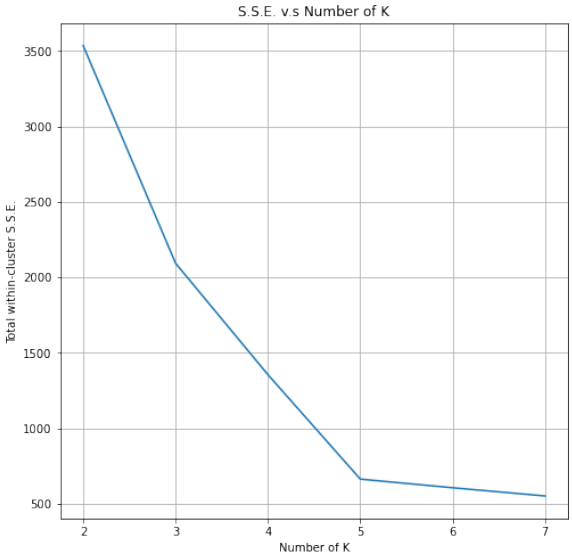


Figure 2: SSE v.s Number of K

within-cluster SSE with the given information. You should first find out the centroid that minimizes the SSE (We're using Euclidean distance). Please refer to Figure 4.

$$A(1,1)$$
$$B(3,3)$$
$$C(5,1)$$
$$D(9,2)$$

The detection of outliers (points that should be put into another cluster) is another challenge in clustering problems, and a naive way to identify the outlier is to remove one of the points and then observe the change of within-cluster SSE. Repeat this step until the outlier that leads to the largest drop in SSE is confirmed.

Assume we know there is an outlier in the given cluster, identify it, and calculate the new minimum within-cluster SSE. Then, explain why you choose this point as the outlier.

1.4 ① First, we choose point A (1,1) as the outlier
Then the centroid of the other 3 points B(3,3), C(5,1), D(9,2)
should be p(5.67, 2) . SSE = 20.67

② Secondly, we choose point B(3,3) as the outlier
Then the centroid of the other 3 points A(1,1), C(5,1), D(9,2)
should be p(5, 1.33), SSE = 32.67

③ Thirdly, we choose point C(5,1) as the outlier
Then the centroid of the other 3 points A(1,1), B(3,3), D(9,2)
should be (4.33, 2), SSE = 36.67

④ Fourthly, we choose point D(9,2) as the outlier
Then the centroid of the other 3 points A(1,1), B(3,3), C(5,1)
should be (3, 1.67), SSE = 10.67

Since SSE = 10.67 is the smallest among the 4 situations, the point D is the outlier

Figure 3: Enter Caption

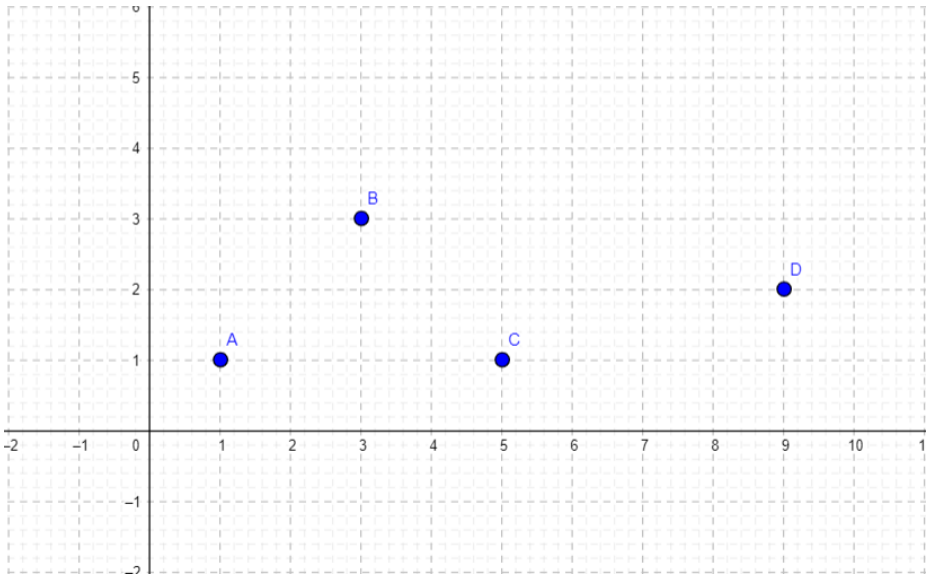


Figure 4: Points Within a Cluster

Question 2: Linear Classifiers, SVM and Mapping Trick [21 pts]

1. **Linear Classifiers [4 pts]**
   Consider two classification problems in a 2D space:

(a) Data points belonging to label `'0'`: {(0,0), (0,1)}.
Data points belonging to label `'1'`: {(1,0), (1,1)}.

(b) Data points belonging to label `'0'`: {(0,0), (1,1)}.
Data points belonging to label `'1'`: {(1,0), (0,1)}.

Can linear classifiers learn each pattern in these two classification problems? Justify your answer.
(a) For the first classification problem:

- Data points belonging to label '0': {(0,0), (0,1)}
- Data points belonging to label '1': {(1,0), (1,1)}

In this case, the two classes are linearly separable. A linear classifier, such as a linear support vector machine (SVM) or logistic regression, can learn to classify the data points correctly by finding a linear decision boundary that separates the two classes. For example, a decision boundary could be the line x = 0.5, which separates the points (0,0) and (0,1) from the points (1,0) and (1,1). So, a linear classifier can learn the pattern as the classes are linearly separable.
(b) For the second classification problem:

- Data points belonging to label '0': {(0,0), (1,1)}
- Data points belonging to label '1': {(1,0), (0,1)}

In this case, the two classes are not linearly separable. No matter how we draw a straight line to separate the two classes, there will always be at least one misclassified point. Therefore, a linear classifier cannot learn to classify the data points correctly for this problem. So, a linear classifier cannot learn the pattern as the classes are not linearly separable.

2. **1-D Linear Classification [5 pts]**
To demonstrate the mapping trick as we discussed in the lecture, we first consider a 1D classification problem. Suppose that we have the following points to be classified:

Class A: (-2, -1.8, 0.3, 0.6, 1) Label: -1
Class B: (-1, -0.7, -0.6, -0.3, 2) Label: 1

Plot the points on a number axis, and comment on whether they can be perfectly classified with a 1D linear classifier.
Hint: a linear classifier in 1D can be seen as assigning different labels to $x > b$ and $x < b$ where b is the division (hyperplane).

3. **The Mapping Trick [6 pts]**

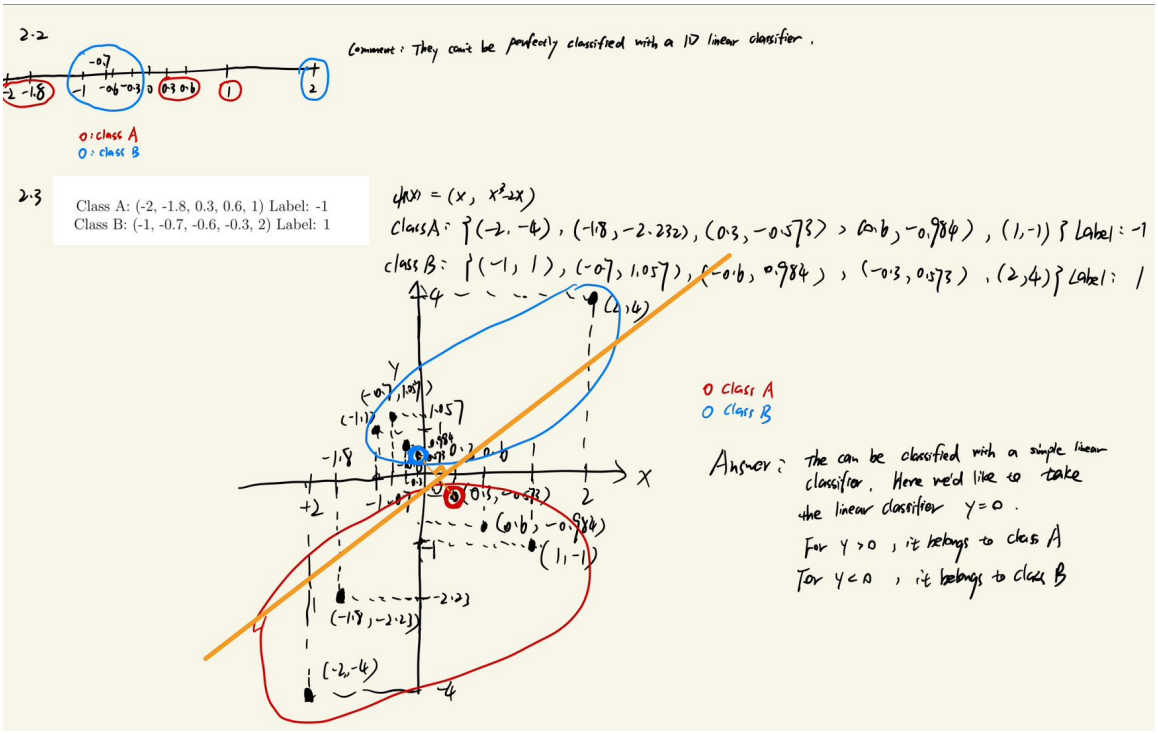


Figure 5: Plots for Q2.2 and 2.3

Now, transform the points into a 2D space using the following mapping function:

$$\psi(x) = (x, x^3 - 2x)$$

Again, plot the points in a 2D coordinate system and comment on whether they can be classified with a simple linear classifier and explain why or why not. Make sure the scales of your x-axis and y-axis are the same; it will make your next questions much easier.

4. **SVM and hyperplane [6 pts]**
Based on the plot that you have obtained from the previous questions, follow the steps below and find the hyperplane for the SVM. (You don't need to solve for the line algebraically.)

   (a) What is the geometric interpretation of the hyperplane in SVM?
   (a) The hyperplane in Support Vector Machine (SVM) serves as the decision boundary that separates classes in a dataset. Geometrically, in a two-dimensional space, the hyperplane is a line. In three dimensions, it becomes a plane. However, in higher dimensions, where SVMs are typically applied, it's challenging to visualize, but it's a multidimensional surface. This hyperplane is positioned to maximize the margin between the classes,

ensuring the largest possible separation between the nearest points of different classes, known as support vectors. The margin represents the perpendicular distance from the hyperplane to the nearest data point of each class. SVM aims to find the hyperplane that best separates the data while minimizing classification errors, making it an effective tool for classification tasks in machine learning.

(b) Circle the point in each class that is the closest to the other class. What is the relationship between these two points and the hyperplane?
(b) As shown in the figure. The points in each class that are closest to the other class are known as the support vectors. These support vectors define the margin of the hyperplane in Support Vector Machine (SVM). Geometrically, the distance from each support vector to the hyperplane is equal and represents the margin of separation between the two classes. The hyperplane is positioned equidistant from these support vectors, maximizing the margin and ensuring the optimal separation between classes. Therefore, the relationship between these two points and the hyperplane is that they define the margin and play a crucial role in determining the decision boundary in SVM.

(c) Based on the above relationship, describe how to find the hyperplane (which should simply be a straight line) and draw it in the plot. (c) The hyperplane is drawn in yellow.

1. Calculate the middle point between the two support vectors. This point will lie on the hyperplane.

2. Determine the slope of the line passing through the middle point and perpendicular to the line connecting the support vectors. The slope of the perpendicular line is the negative reciprocal of the slope of the line connecting the support vectors.

3. Use the middle point and the slope to determine the equation of the hyperplane (straight line) using the point-slope form of a line equation: y - y = m(x - x), where (x, y) is the middle point and m is the slope.

4. Draw the hyperplane (straight line) on the plot, ensuring it passes through the middle point and is perpendicular to the line connecting the support vectors.

Question 3: Neural Network Basics [20 pts]

1. For a classification problem with 16 input features and 16 classes, compare the following two networks:

   (a) a deeper network with nine dense (fully connected) layers with 32 nodes in each layer

   (b) a shallower network that has three dense (fully connected) layers with 128 nodes in each layer

   If we use `float32` to store the weights, what are the memory requirements of these two configurations? Which one would you prefer to deploy on an edge device? (The networks have no bias. 32 bits is 4 bytes. The network in Slide 4 of Lecture 9 has four fully connected layers.) **[4 pts]**
   Hint: The input layer is not included in the above configuration.
   (1)Calculate the number of parameters (weights) in each network:
   (a) Deeper network:

   - Number of layers = 9
   - Nodes in each layer = 32
   - Number of weights per layer = (Number of nodes in current layer) × (Number of nodes in previous layer). Total number of weights = 16×32+32 × 32 × 8 +32×16= 9216.

   (b) Shallower network:

   - Number of layers = 3
   - Nodes in each layer = 128
   - Number of weights per layer = (Number of nodes in current layer) × (Number of nodes in previous layer). Total number of weights= 16×128+128 × 128×2+128×16 = 36864.

   (2) Calculate the memory requirements for storing these parameters:
   (a) Deeper network:

   - Memory requirements = Total parameters × Size of each parameter
   - Memory requirements = 9216 parameters × 4 bytes (since each parameter is stored using float32) = 36864 bytes

   (b) Shallower network:

   - Memory requirements = Total parameters × Size of each parameter

- Memory requirements = 36864 parameters × 4 bytes = 147456 bytes.

Comparing the memory requirements:

- Deeper network: 36,864 bytes
- Shallower network: 147,456 bytes

In the comparison provided, the shallower network (option b) has a larger memory requirement compared to the deeper network (option a). This is because the shallower network has larger layer sizes (128 nodes in each layer) compared to the deeper network (32 nodes in each layer). As a result, it requires more memory to store the parameters.

Therefore, in the context of deploying on an edge device with limited memory resources, it would be preferable to choose the option with lower memory requirements to ensure efficient utilization of resources. Therefore, the deeper network with nine layers (option a) would be the better choice for deployment on an edge device due to its lower memory requirement despite having fewer nodes in each layer.

2. You have trained a DNN model for static image classification applications. You deploy the model in the field with a camera. However, you find that the camera's view has shifted in space compared to the camera used for capturing training images. In other words, the area of interest has moved off-center while the objects in your training images are perfectly at the center. What would happen to the classification accuracy if you used (1) a Multi-layer Perceptron or (2) a Convolutional Neural Network as your model? Justify your answer (Think about Translation Invariance).
   What would happen to the performance of the model if you add max pooling layers into CNN in this scenario? **[3 pts]**

   (a) Multi-layer Perceptron (MLP): MLPs are not inherently equipped to handle translation invariance. Translation invariance refers to the ability of a model to recognize objects regardless of their position within the input data. Since MLPs treat each input feature independently and do not consider spatial relationships, they are highly sensitive to changes in the position of the input data. Therefore, when the camera's view shifts, causing the area of interest to move off-center, the classification accuracy of an MLP is likely to decrease significantly. The model would struggle to correctly classify objects that are no longer centered in the input images.

   (b) Convolutional Neural Network (CNN): CNNs, on the other hand, are designed to capture spatial hierarchies and exhibit some degree of translation invariance through their convolutional layers. Convolutional layers apply filters across different regions of the input image, allowing the network to

learn spatial patterns and features regardless of their exact position within the image. As a result, CNNs are more robust to changes in the position of objects within the input data compared to MLPs. Therefore, while the classification accuracy of a CNN may still be affected by the shift in the camera's view, it is likely to maintain higher accuracy compared to an MLP in the same scenario.

Regarding the addition of max pooling layers into the CNN: Max pooling layers are often used in CNN architectures to downsample the spatial dimensions of the feature maps, reducing computational complexity and providing a form of spatial invariance. However, adding max pooling layers could potentially exacerbate the issue of off-center objects in this scenario. Max pooling layers typically select the maximum value within each pooling region, discarding the positional information of features. If the area of interest has shifted off-center, max pooling could lead to the loss of important spatial information, further impacting the model's performance in accurately classifying objects. Therefore, in this specific scenario, adding max pooling layers may not necessarily improve the performance of the model and could potentially worsen the impact of the off-center shift in the camera's view.

3. Batch normalization layers are used in most of today's state-of-the-art convolutional neural networks. We can *fuse* the batch normalization layer into the convolutional layer. Suppose we have w_i and b_i to denote the weight and bias in the convolutional layer. Write down the new w_i' and b_i' for the fused convolutional layer with the parameters from the batch normalization. **[4 pts]**
After fusing batch normalization into the convolutional layer, the new parameters w_i' and b_i' can be expressed as follows:
Certainly, let's modify the notation as requested:
1. **Effective weights (w_i')**: The effective weights are obtained by scaling the original convolution weights (w_i) with the gamma parameter (γ) from the batch normalization layer, divided by the square root of the variance (σ) plus a small epsilon (ϵ) for numerical stability:

$$w_i' = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \cdot w_i$$

2. **Effective bias (b_i')**: The effective bias is calculated by multiplying the original convolution bias (b_i) with the gamma parameter (γ), then subtracting the product of the mean (μ) and the effective weights (w_i'), and finally adding the beta parameter (β) from the batch normalization layer:

$$b_i' = \gamma \cdot \left(b_i - \frac{\mu}{\sqrt{\sigma^2 + \epsilon}} \cdot w_i \right) + \beta$$

- w_i: Original weights for the i-th convolutional filter.
- w_i': Effective weights for the i-th convolutional filter.
- γ: Scaling factor learned by batch normalization.
- σ: Variance of the mini-batch learned by batch normalization. - ϵ: Small constant (epsilon) for numerical stability.
- μ: Mean of the mini-batch learned by batch normalization.
- b_i': Effective bias for the i-th convolutional filter.
- b_i: Original bias for the i-th convolutional filter.

4. You are building a classifier and considering three different activation functions: ReLU, tanh, and unit step function.

   (a) Please sketch the three activation functions. **[3 pts]**

   (b) Which one is the most popular of the three? Name two advantages of this activation function over the other two. **[2 pts]**

(a) Sketches of the three activation functions:
1. **ReLU (Rectified Linear Unit)**: - ReLU is defined as $f(x) = \max(0, x)$. - It is linear for positive values of x and zero for negative values of x. - ReLU is widely used in deep learning due to its simplicity and effectiveness in combating the vanishing gradient problem.
2. **tanh (Hyperbolic Tangent)**:
- Tanh is defined as $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$.

- It squashes the input values into the range $(-1, 1)$, making it suitable for classification tasks where the output needs to be bounded.
- Tanh is symmetric around the origin and has steeper gradients compared to sigmoid, making it less susceptible to the vanishing gradient problem.

3. **Unit Step Function**:
- Unit Step Function is defined as $f(x) = \begin{cases} 0, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases}$.
- It outputs 0 for negative inputs and 1 for non-negative inputs.
- The unit step function is rarely used in modern neural networks due to its discontinuity and lack of gradient information, which makes it unsuitable for training using gradient-based optimization algorithms.
Sketches:

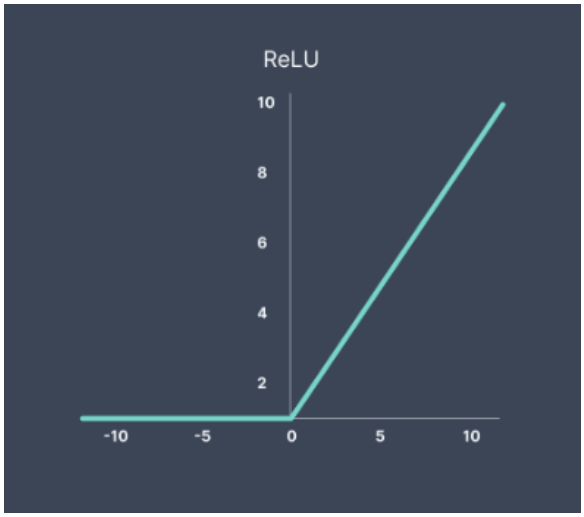


Figure 6: ReLu Function

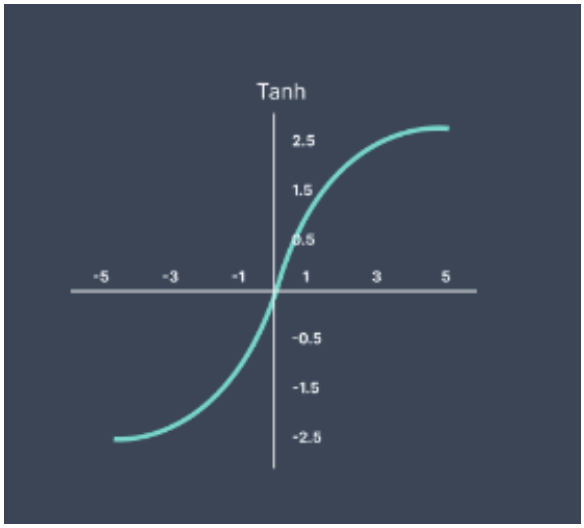


Figure 7: tanh Function

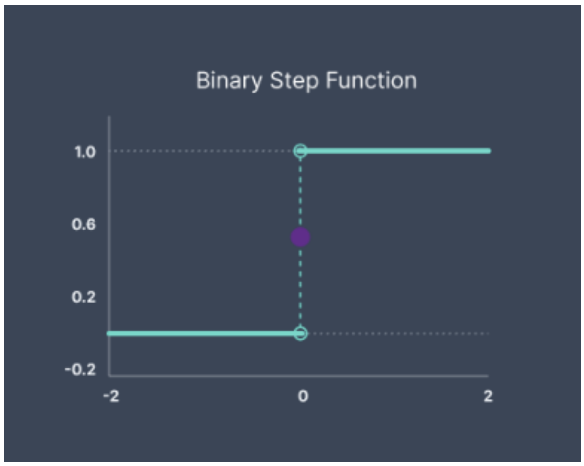


Figure 8: Unit Step Function

(b) The most popular activation function of the three is ReLU (Rectified Linear Unit). Two advantages of ReLU over tanh and the unit step function are:

1. **Sparsity**: ReLU introduces sparsity by setting negative values to zero. This sparsity can help prevent overfitting by reducing the number of active neurons, thereby promoting more efficient representations in the network.

2. **Efficiency**: ReLU is computationally more efficient compared to tanh and sigmoid functions because it involves simple thresholding operations without expensive exponential calculations. This efficiency contributes to faster training and inference times, especially in deep neural networks with many layers.

5. Assuming your design needs to fetch N weights from the memory and the average memory access bandwidth is B bytes per second, calculate the data transfer latency (in the unit of second) when using FLOAT32 to represent each weight. Then, calculate the data transfer latency if we apply quantization to convert the model to INT8. **[4 pts]**

Let's denote: - N: Number of weights
- W_{size}: Size of each weight in bytes (for FLOAT32, $W_{\text{size}} = 4$ bytes)
- B: Average memory access bandwidth in bytes per second

Then, the data transfer latency (T_{latency}) in seconds can be calculated as:

$$T_{\text{latency}} = \frac{N \cdot W_{\text{size}}}{B}$$

Now, let's calculate the data transfer latency for both FLOAT32 and INT8 representations:

1. **For FLOAT32**:
- Given N weights and $W_{\text{size}} = 4$ bytes for FLOAT32:

$$T_{\text{latency\_FLOAT32}} = \frac{N \cdot 4}{B}$$

2. **For INT8**: - When quantizing the model to INT8, the size of each weight is reduced to 1 byte.
- Thus, $W_{\text{size}} = 1$ byte for INT8.
- Therefore, the data transfer latency for INT8 ($T_{\text{latency\_INT8}}$) can be calculated as:

$$T_{\text{latency\_INT8}} = \frac{N \cdot 1}{B} = \frac{N}{B}$$

Question 4: Convolution [16 pts]

1. We want to perform a convolution operation on a feature map. This function takes the feature map and the kernel as inputs and returns the output features. Calculate and fill in the parentheses with proper numbers of the dimensions of the output feature map and loop ranges. Assume there is no padding to the input feature map.[**6 pts**]

```
def convolution(input_fm, kernel) {
    """
    Inputs:
    - input_fm:   A 3D numpy array of shape
                  (3, 223, 223).
    - weights:    A 4D numpy array of shape
                  (16, 3, 3, 3).

    Returns:
    - output_fm:  A 3D numpy array of shape
                  (channel_out, height_out, width_out).
    """

    height_out = (    221        ) #height of the output feature map
    width_out = (    221        ) #width of the output feature map
    channel_out = (    16        ) #channel number of the output feature map

    output_fm = np.zeros((channel_out, height_out, width_out))
    #empty feature map in the designated shape

    for row in range(   221         ):
        for col in range(     221       ):
            for to in range(    16       ):
                for ti in range(   3       ):
                    for ki in range(      3      ):
                        for kj in range(     3       ):
                            output_fm[to][row][col] +=
        weights[to][ti][ki][kj] * input_fm[ti][row + ki][col + kj]

    return output_fm
}
```

Figure 9: Q4.1 Stride 1 Convolution

2. If we wish to perform a stride-2 convolution instead of a stride-1 (unit stride) convolution such as the one above, which numbers will change? Please fill in the new numbers in the blanks below. Again, no padding for the input feature map.[**2 pts**]

```
def convolution_s2(input_fm, kernel) {
    """
    Inputs:
    - input_fm:   A 3D numpy array of shape
                    (3, 223, 223).
    - weights:    A 4D numpy array of shape
                    (16, 3, 3, 3).

    Returns:
    - output_fm:  A 3D numpy array of shape
                    (channel_out, height_out, width_out).
    """

    height_out = (    111        )
    output_out = (   111         )
    channel_out = (    16        )

    output_fm = np.zeros((channel_out, height_out, width_out))

    for row in range(      111      ):
        for col in range(      111      ):
            for to in range(    16        ):
                for ti in range(     3       ):
                    for ki in range(     3       ):
                        for kj in range(     3       ):
                            output_fm[to][row][col] +=
        weights[to][ti][ki][kj] * input_fm[ti][row + ki][col + kj]

    return output_fm
}
```

Figure 10: Q4.2 Stride 2 Convolution

3. Consider the first question with the normal unit-stride convolution; we use that as one layer in our neural network. Please think about the following questions and answer in one to two sentences.

   (a) Can the loops that go through the kernel dimension (ki, kj) be swapped with the ones that go through the output dimensions (row, col) without causing any issues? Explain when it's okay to switch the order of loops and

how it might affect the speed and way of accessing memory. **[2 pts]**

(a) Swapping the loops that iterate through the kernel dimensions (ki, kj) with those iterating through the output dimensions (row, col) can lead to incorrect results in the convolution operation. This is because the order of loops determines the sequence in which elements are accessed in memory. In a typical convolution operation, the kernel elements are multiplied with corresponding input feature map elements at different spatial locations, and then these products are summed to produce the output feature map. If the loop order is changed, the memory access pattern is altered, leading to incorrect calculations and potentially invalid outputs.

Additionally, changing the loop order can impact the efficiency of memory access and computational performance. Modern computer architectures, including CPUs and GPUs, exploit data locality and caching mechanisms to optimize memory access patterns. By adhering to the natural order of loops (kernel dimensions first, followed by output dimensions), the memory access pattern aligns with the cache hierarchy, improving data reuse and overall performance. Swapping the loop order may disrupt this optimization and result in slower execution due to increased cache misses and inefficient memory access patterns.

(b) If you wish to increase the **receptive field** of your network layer, what do you need to change? **[2 pts]**

(b) To increase the receptive field of a network layer, one typically needs to adjust the size of the convolutional kernel/filter used in the layer. The receptive field refers to the spatial extent of input data that influences the computation of a particular output feature map element. By using larger kernels, the network can capture more extensive contextual information from the input data, allowing it to learn more complex patterns and relationships.

(c) What is one potential benefit of a larger receptive field? Can a larger receptive field do any harm in practical applications? **[2 pts]**

(c)One potential benefit of a larger receptive field is the ability to capture more contextual information and spatial dependencies within the input data. This can lead to enhanced feature extraction and improved performance in tasks such as object recognition, where understanding the context of objects in an image is crucial for accurate classification or localization.

However, there are trade-offs to consider when increasing the receptive field. A larger receptive field typically entails using larger convolutional kernels, which increases the number of parameters in the network and

computational complexity. This can lead to higher memory requirements and computational costs during training and inference. In practical applications, especially those with constrained computational resources or real-time processing requirements, the trade-off between increased receptive field and computational efficiency must be carefully considered. Additionally, excessively large receptive fields may introduce issues such as overfitting to specific training data or increased susceptibility to noise in the input data.

(d) What is the significance of the output channel and how is the number of output channels determined? Why is it usually larger than the input channel? **[2 pts]**

(d) The output channel in a convolutional layer signifies the number of filters or feature maps produced by that layer. Each filter learns to extract specific features or patterns from the input data, leading to a diverse set of feature maps in the output. The number of output channels is determined by the design choices of the network architect (mainly by the number of number of filters in that layer) and is typically larger than the number of input channels.

Having a larger number of output channels allows the network to learn a richer set of features and representations from the input data. This increased capacity for feature extraction enables the network to capture more complex patterns and relationships within the data, enhancing its ability to generalize and make accurate predictions on unseen examples.

Furthermore, increasing the number of output channels provides the network with greater flexibility to learn diverse representations of the input data, which can be beneficial for tasks requiring nuanced feature extraction, such as image classification or semantic segmentation.

Question 5: Backpropagation [15 pts]

Consider a simple neural network in Figure 11. Single-circled nodes denote variables: x_1 is an input variable, h_1 and h_2 are intermediate variables (first layer), and $\hat{y}$ is an output variable (second layer), etc. Double-circled nodes denote functions: Σ takes the sum of its inputs, and σ denotes the logistic function $\sigma(x) = \frac{1}{1+e^{-x}}$.

Suppose we have an MSE (Mean Square Error) loss $L(y, \hat{y}) = \|y - \hat{y}\|_2^2$, We are given a data point $(x1, x2, x3, x4) = (0.3, 1.4, 0.9, -0.6)$ with the true label 0.37. The gradient of the MSE loss function is $2\|y - \hat{y}\|$.

1. First perform forward propagation and compute s_1, s_2, s_3, h_1, h_2, and $\hat{y}$. Then, use the backpropagation algorithm introduced in Lecture 8 to compute the partial derivative $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial w_3}, \frac{\partial L}{\partial w_4}, \frac{\partial L}{\partial w_5}, \frac{\partial L}{\partial w_6}$. **[12 pts]**

   Hint: Remember that during backpropagation, the gradient computed for a later layer can be reused when calculating the gradient for earlier layers.
   s1=x1* w1+x2*w2=-0.23
   s2=x3*w3+x4*w4=-1.92
   s3=h1*w5+h2*w6=0.2352
   h1=0.4428
   h2=0.1279
   $\hat{y}$ =$\sigma(h1 * w5 + h2 * w6)$=0.5585
   $\frac{\partial L}{\partial w_1}$=0.0021,$\frac{\partial L}{\partial w_2}$=0.0100,$\frac{\partial L}{\partial w_3}$=0.0075,$\frac{\partial L}{\partial w_4}$=-0.005,$\frac{\partial L}{\partial w_5}$=0.0412,$\frac{\partial L}{\partial w_6}$=0.0119
2. Explain vanishing gradients and exploding gradients. You can watch this video before answering the question. **[3 pts]**

(1)Vanishing Gradients: Vanishing gradients refer to the problem where gradients become extremely small as they are backpropagated through many layers of a deep neural network during the training process. This phenomenon is more likely to occur in networks with many layers or with certain activation functions that have gradients close to zero in certain regions, such as the sigmoid function.
When gradients vanish, it means that the updates to the network's weights become insignificant, leading to very slow or no learning in deeper layers of the network. As a result, these layers fail to effectively learn meaningful representations from the data, and the overall performance of the network may suffer.

One common cause of vanishing gradients is the use of activation functions like the logistic, sigmoid or tanh functions, which saturate to very small values for large positive or negative inputs. Another cause can be poor weight initialization or excessively deep networks, where gradients diminish as they are backpropagated through numerous layers.

(2)Exploding Gradients: On the other hand, exploding gradients occur when gradients become extremely large during training, often leading to numerical instability. This can happen when the gradient magnitudes grow exponentially as they are backpropagated through many layers. As a result, weight updates become extremely large, causing the model to diverge rather than converge to a solution.

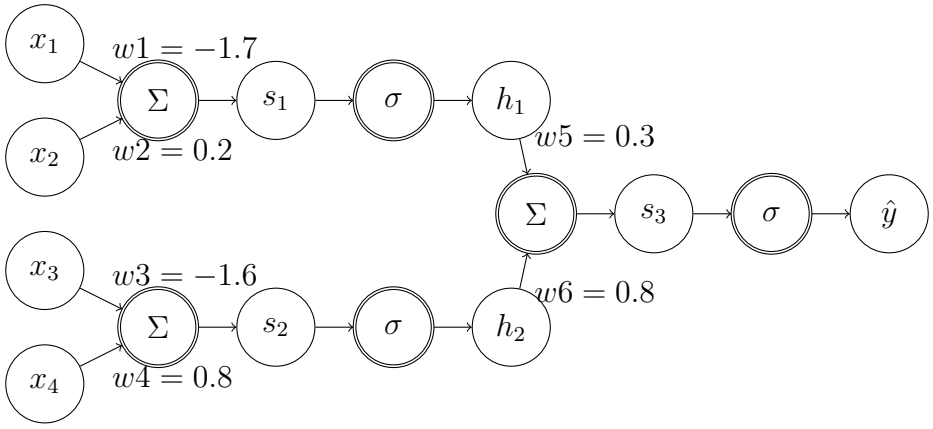


Figure 11: A Simplified Neural Network Example

Question 6: Find Bugs in Tensorflow Code [8 pts]

Bob has recently working on Lab 2. He is not satisfied with small datasets such as Fashion MNIST, and he wants to explore more interesting datasets. He decides to build a simple Multi-Layer Perceptron (MLP) classifier using TensorFlow with Keras to classify images from the CIFAR-10 dataset. The CIFAR-10 dataset consists of 60,000 32x32 color images in 10 classes, with 6,000 images per class, which is a cool choice for developing and testing image classification models.

However, Bob is relatively new to TensorFlow and neural network design. He wrote the following code to build, compile, and train his MLP model. However, Bob noticed that his model doesn't work, and he suspects there must be some bugs in his code.

As Bob's helpful friend, you would like to review Bob's code, identify any bugs that might be present, and **explain** how they could affect the model's performance by making a short comment next to the code. After identifying the bugs, suggest corrections to make the model work. (Hints: There are 4 bugs in the code. For information on possible loss functions, please refer to the link.)

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical

# Load and preprocess the CIFAR-10 dataset as int arrays
(train_images, train_labels), (test_images, test_labels) \
        = cifar10.load_data()
# Normalize the images
train_images, test_images \
        = train_images * 255.0, test_images * 255.0
train_labels, test_labels \
        = to_categorical(train_labels), to_categorical(test_labels)

# Build the MLP model
model = models.Sequential([
    layers.Flatten(input_shape=(32, 32, 1)),
    layers.Dense(512, activation='relu'),
    layers.Dense(256, activation='relu'),
    layers.Dense(10, activation='relu')
])

# Compile the model
model.compile(optimizer='adam',
```

```
25                 loss='binary_crossentropy',
26                 metrics=['accuracy'])
27
28 # Train the model
29 model.fit(train_images, train_labels, epochs=10, \
30         batch_size=64, validation_split=0.2)
31
32 # Evaluate the model
33 test_loss, test_acc = model.evaluate(test_images, test_labels)
34 print(f'Test accuracy: {test_acc:.3f}')
```

1. Input Shape Mismatch:
   - Bug: The input shape specified in the first layer of the model (`layers.Flatten(input_shape=(32, 32, 1))`) does not match the actual shape of the CIFAR-10 images, which are 32x32 color images.
   - Impact: This will cause a shape mismatch error during model training.
   - Correction: Update the input shape to `(32, 32, 3)` to match the dimensions of color images in CIFAR-10 dataset.
2. Activation Function in Output Layer:
   - Bug: The output layer uses the ReLU activation function (`activation='relu'`). For multi-class classification tasks like CIFAR-10, the output layer should use the softmax activation function to output probabilities for each class.
   - Impact: Using ReLU activation in the output layer is incorrect for multi-class classification and will lead to incorrect predictions.
   - Correction: Change `activation='relu'` to `activation='softmax'` for the output layer.
3. Loss Function Mismatch:
   - Bug: The loss function specified in `model.compile()` is `'binary_crossentropy'`, which is suitable for binary classification tasks but not for multi-class classification tasks like CIFAR-10.
   - Impact: Using the wrong loss function will mislead the optimization process and hinder model performance.
   - Correction: Change `loss='binary_crossentropy'` to `loss='sparse_categorical_crossentropy'` since the labels are integers in CIFAR-10 dataset.
4. Normalization Issue:
   - Bug: The normalization of image data (`train_images` and `test_images`)

is not performed correctly. Multiplying by `255.0` is likely intended for scaling the pixel values to the range [0, 1], but it should be divided by `255.0` instead of multiplied.

- Impact: Incorrect normalization can significantly affect model convergence and performance.
- Correction: Change
  `train_images, test_images = train_images * 255.0, test_images * 255.0`
  to `train_images, test_images = train_images / 255.0, test_images / 255.0`
  for correct normalization.