CSCI933 Machine Learning: Algorithms and Applications

Karan G: 7836685

Final Exam
June 18, 2024

1 Question 1

(a) Designing a Deep Neural Network Classifier for Wound Healing Stages

To design a deep neural network classifier with the constraint of insufficient example data, we need to enhance generalization using techniques like regularization. We also need to ensure that our classifier can accurately predict each wound stage by employing a deep network architecture capable of effective feature extraction.

Given these conditions, I will use a modified ResNet architecture, known for its depth and ability to extract complex features. ResNet's residual connections help maintain effective gradient flow, enabling the network to learn better and generalize well.

The network starts with an input layer handling wound images of size 224x224 pixels with three color channels (RGB). The initial layers perform convolution and batch normalization to extract low-level features and stabilize training. Max pooling is used to reduce the spatial dimensions and distill significant features.

We use multiple convolutional layers: the first with 32 filters, the second with 64 filters, and the third with 128 filters, all using a 3x3 kernel size and ReLU activation. Each convolutional layer is followed by batch normalization to stabilize and speed up training, and max pooling with a 2x2 filter size to reduce spatial dimensions and distill significant features.

Following these layers, residual blocks with convolutional layers and skip connections help the network learn residual mappings, making optimization easier. Each block has two convolutional layers with batch normalization and ReLU activation, while skip connections ensure gradient flow and better feature learning.

Next, a Global Average Pooling layer is added, followed by fully connected layers to integrate and interpret these high-level features.

The final output layer is a dense layer with four units, corresponding to the four stages of wound healing, and a softmax activation function to provide a probability distribution over the stages.

We can use categorical cross-entropy as the loss function, suitable for multi-class classification problems, and the Adam optimizer for its efficiency.

(b) Variational Autoencoder (VAE) for Data Augmentation

To address the lack of sufficient data for training a deep neural network classifier for wound image classification, I can employ a variational autoencoder (VAE) to generate more wound image data. A VAE is a type of generative model that can learn the underlying distribution of the input data and produce new, similar samples, thus augmenting our dataset.

The design of a suitable VAE for generating wound image data includes three key subsystems: the encoder, the latent space sampling mechanism, and the decoder.

The encoder's job is to map the input images into a latent space. This process involves passing the wound images through several convolutional layers that capture the essential features of the images. Each convolutional layer uses filters and ReLU activation, followed by batch normalization to stabilize training and max pooling to reduce the spatial dimensions. These steps ensure that only the most significant features are retained. The final output of the encoder is then flattened and passed through fully connected layers that produce two outputs: the mean and the log variance of the latent space.

Next, we use these outputs to sample a point in the latent space. This sampling uses a reparameterization trick to ensure the model can learn and generate continuous variations of the input data. Essentially, this step involves creating a new representation by combining the mean and variance with a random component.

The decoder takes this sampled point and maps it back to the original image space. It mirrors the encoder's process but in reverse. Starting with fully connected layers, the decoder reshapes the latent vector back into a format suitable for upsampling. The subsequent deconvolutional layers use filters and ReLU activation, followed by batch normalization and upsampling, to progressively reconstruct the image, layer by layer. The final output layer uses a sigmoid activation function to produce the reconstructed image.

2 Question 2

(a) Filter Equation

Analysis of the Filter Equation with Given Input

With the provided parameters $x[n] = [0.5, 0.4, 0.7, 0.5, 0.6, 0.4, 0.8, 0.5]$, $C = 0.1$, $N = 2$, $y[-1] = 0.0$, and $y[-2] = 0.0$, we will calculate $y[n]$ for $n = 0$ to $n = 5$.

For $n = 0$, $y[0] = y[-1] + y[-2] + x[0] + C = 0.0 + 0.0 + 0.5 + 0.1 = 0.6$. For $n = 1$, $y[1] = y[0] + y[-1] + x[1] + C = 0.6 + 0.0 + 0.4 + 0.1 = 1.1$.

For $n = 2$, $y[2] = y[1] + y[0] + x[2] + C = 1.1 + 0.6 + 0.7 + 0.1 = 2.5$.

For $n = 3$, $y[3] = y[2] + y[1] + x[3] + C = 2.5 + 1.1 + 0.5 + 0.1 = 4.2$.

For $n = 4$, $y[4] = y[3] + y[2] + x[4] + C = 4.2 + 2.5 + 0.6 + 0.1 = 7.4$.

For $n = 5$, $y[5] = y[4] + y[3] + x[5] + C = 7.4 + 4.2 + 0.4 + 0.1 = 12.1$.

This shows how the filter uses past values to influence the current output, thereby achieving memory.

Comparison with LSTM Cell

The given filter functions similarly to an LSTM cell in that both use past information to inform the current output, creating a form of memory.

An LSTM cell achieves memory through its structure which includes a forget gate that decides what information to discard from the cell state, an input gate that updates the cell state with new information, and an output gate that determines what the next hidden state should be. The memory mechanism in LSTM is more sophisticated with a long-term state (c) that keeps track of long-term dependencies and a short-term state (h) that reflects the immediate information being processed. LSTM cells can learn to retain important information and discard irrelevant details over long sequences.

Thus, while both the filter and LSTM cell use past outputs to affect current outputs (memory), LSTMs achieve this with more flexibility and capability to handle long-term dependencies through their specialized gates.

(b) Image Captioning System Using a Transformer Model

To design an image captioning system using a transformer model, the task can be treated as neural machine translation, where the source is the image and the target is the caption. Here's a detailed design of the system:

1. Image Feature Extraction: Use a convolutional neural network (CNN) like ResNet, GoogleNet to extract features from the input image, producing a high-dimensional feature map. Transform raw image pixels into a structured representation encoding relevant visual information.

2. Transformer Encoder: Convert the 2D feature map into a sequence of vectors by flattening the spatial dimensions. Feed this sequence into the transformer encoder to capture complex relationships within the image data. Encode image feature vectors into high-level representations using multi-head self-attention and feed-forward neural networks.

3. Positional Encoding: Add positional encoding to the image feature vectors to provide information about the position of each vector in the sequence. Inject positional information to maintain the spatial structure of the original image.

4. Transformer Decoder: The decoder generates the caption word by word. During training, it takes the encoder's output and the target sequence shifted by one position. It attends to the encoded image features and previously generated words to predict the next word.

5. Attention Mechanism: The decoder uses attention to focus on different parts of the image when generating each word, computing a weighted sum of the encoder outputs. This Enable the decoder to focus on relevant parts of the image for generating each word in the caption.

6. Output Layer and Softmax: The final layer of the decoder is a dense layer followed by a softmax activation function, mapping the decoder's hidden states to a probability distribution over the vocabulary.

7. Training Process: Train the model end-to-end using a dataset of images paired with captions, using categorical cross-entropy loss.

8. Inference Process: During inference, generate captions by sampling or selecting the next word based on the highest probability until an end-of-sequence token is generated. Use beam search to improve the quality of the generated captions.

3 Question 3

(a) Fruit-nutrition-picker suggester

To design a fruit-nutrition-picker application using a Graph Convolutional Network (GCN), I can start by constructing a "world model" that accurately represents fruits and their nutritional properties. This model is based on graph theory, where we define a graph $G = (V, E)$, with V representing the set of fruits and E representing the relationships between these fruits based on their nutritional similarities and differences.

World Model

Each node in our graph V corresponds to a specific fruit, and the features of each node include nutritional properties such as calories, vitamins, antioxidants, proteins, and carbohydrates. These features form the attribute vectors for the nodes, encapsulating the nutritional profile of each fruit. The edges E between the nodes capture the relationships between the fruits, which can be determined based on similarity in nutritional content or other relevant criteria.

Graph Construction

The graph can be represented using an adjacency matrix A, where $A_{ij} = 1$ if there is an edge between fruit i and fruit j, and $A_{ij} = 0$ otherwise. Additionally, a weight matrix W can be used indicating the strength of the relationship such as the degree of similarity in nutritional properties.

GCN Architecture

The architecture involves multiple GCN layers, each performing a localized, weighted aggregation of feature information from neighboring nodes.

The final layer outputs a representation for each node that captures both its own features and the aggregated features of its neighbors.

Model Training

The model is trained using labeled data where each fruit combination is associated with one of the nutritional goals. The training process involves minimizing a loss function, like cross-entropy loss or negative likelihood loss, over the training dataset. The dataset is divided into training and validation sets to monitor performance and prevent overfitting. The training process includes initialization of the GCN parameters, a forward pass through the GCN for each training example to compute the predicted output, calculation of the loss, backpropagation to compute the gradients of the loss with respect to the GCN parameters, and updating the GCN parameters using an optimization algorithm such as stochastic gradient descent (SGD).

Inference Strategy

During inference, given a new set of fruits, the trained GCN can suggest the optimal combination to achieve the desired nutritional goal. The model utilizes the learned graph structure and node embeddings to make these predictions. By incorporating both the individual properties of the fruits and their relational context within the graph, the GCN can effectively recommend suitable fruit combinations for each nutritional target.

(b) Reservoir computing network vs Kernel vs Overcomplete autoencoders

Kernel methods, such as those used in support vector machines, transform input data into a higher-dimensional space using a predefined kernel function $k(x_i, x_j)$ to compute the inner product in the transformed space. This method relies on the kernel trick to avoid explicit mapping, enabling efficient computation.

Overcomplete autoencoders, in contrast, are neural networks with a hidden layer of higher dimensionality than the input layer. They consist of an encoder function $h = f(x)$ and a decoder function $r = g(h)$ to reconstruct the input, optimizing the reconstruction loss $L(x, g(f(x)))$. Regularization techniques are applied to encourage meaningful feature discovery.

Reservoir computing networks are designed for sequence data, consisting of an input layer, fixed random-weight recurrent layers (the reservoir), and a trainable output layer. Only the output layer weights are updated during training.

In summary, kernel methods use deterministic functions for implicit mapping, overcomplete autoencoders learn representations through trainable weights, and reservoir computing networks use fixed random weights in recurrent layers, training only the output layer.

(c) Human Action Recognition using Reservoir Computing Network

Given the context of the problem, it is clear that we are solving a classification problem: we need to classify a given action recorded in a short video sequence into one of a predefined set of actions (e.g., waving, running, etc.). One effective way of encoding the input video sequence is to transform each frame into skeleton data, where human joints are represented as nodes. This transformation results in a sequence of skeletons representing an action, capturing the dynamic nature of human movements.

Input Layer

Each frame in the video sequence is processed to extract skeleton data, transforming human joints into nodes. This results in a time-series representation of the action, with each timestep corresponding to the skeleton configuration in a frame. The sequence of skeletons forms the input to the reservoir computing network, preserving the temporal dynamics of the actions.

Reservoir Layer

The sequence of skeleton data is fed into a reservoir layer consisting of recurrent neural network (RNN) units. The reservoir captures temporal dependencies and maps the input sequence to a high-dimensional space. The weights of the RNN units in the reservoir are initialized randomly and remain fixed during training. This fixed random mapping helps in capturing complex temporal patterns without the need for extensive training.

Output Layer

The state of the reservoir at each timestep is fed into a regression layer. This layer outputs continuous values that represent the likelihoods of different action classes. To convert the regression output into a classification task, a softmax function is applied to the continuous outputs to obtain probabilities for each action class. The action with the highest probability is selected as the predicted action.

Training Process

During training, only the weights of the output regression layer are updated using backpropagation. The fixed reservoir ensures that the temporal dynamics are captured efficiently while keeping the training computationally lightweight. The regression layer outputs are transformed into classification probabilities using the softmax function, enabling the network to handle the classification task.

Mitigating Overfitting

- **Dropout**: Introduce dropout in the regression layer during training to randomly drop units and prevent co-adaptation of hidden units, enhancing generalization.
- **L2 Regularization**: Add an L2 penalty term to the loss function to constrain the magnitude of the weights in the regression layer, discouraging overly complex models.

Inference Strategy

For a new sequence of skeletons, the reservoir computing network processes the input sequence through the fixed reservoir layers. The state of the reservoir is used by the trained regression layer to produce likelihoods for each action class. The action with the highest probability is chosen as the predicted action.