[CODE DEMO] Capsule Networks

Status	Deep Learning

Introduction to Capsule Networks

Capsules in Capsule Networks offer a nuanced method for interpreting visual data, going beyond the capabilities of standard neurons in typical neural networks. A capsule is essentially a group of neurons that outputs a multi-dimensional vector, encapsulating various properties of a specific part of an image—like its position, size, orientation, and texture. This vector not only determines the likelihood that a particular feature is present but also describes its characteristics within the spatial hierarchy of the image. By enabling the network to understand and preserve these relationships, capsules represent a pivotal step towards more sophisticated and context-aware image recognition.

Representing Relationships Between Parts

Capsule Networks take object recognition a step further than traditional methods by not just identifying objects but also understanding the context within which they exist. By maintaining

spatial hierarchies between features, they edge closer to human-like perception in visual recognition tasks. This is largely due to dynamic routing, a process that emphasizes the importance of spatial relationships and pose information, which are essential for recognizing objects in various positions and conditions.

Moreover, Capsule Networks have a considerable edge over conventional CNNs. They have the capability to recognize objects in different orientations more accurately, discern overlapping figures, and learn effectively from smaller datasets. This efficiency in learning significantly reduces the dependency on large volumes of training data that traditional neural networks often require to perform effectively.

Model Architecture

Capsule Network's architecture have convolutional layers to detect initial features, primary capsules that capture complex feature combinations, digit capsules to represent whole entities, and a decoder to reconstruct and refine the network's understanding.

Encoder

The encoder in a Capsule Network consists of multiple layers, starting with a convolutional layer. It transforms a simple image into a complex multi-dimensional vector that encodes not just the existence but also the attributes of various image features.

First Layer: Convolutional Layer

The initial layer of the encoder is a convolutional layer that detects features such as edges. It applies a kernel across the image to produce a set of feature maps.
The code snippet for convulational layer:

```
class ConvLayer(nn.Module):
    def __init__(self, in_channels=1, out_channels=256):
        super(ConvLayer,self).__init__()
        self.conv=nn.Conv2d(in_channels, out_channels, kernel_size=9, str
ide=1, padding=0)
    def forward(self,x):
        features=F.relu(self.conv(x))
        return features
```

In the code snippet:

- The ConvLayer class is initialized as a subclass of nn.Module.
- The constructor __init__ sets up the convolutional layer with the specified parameters:

    - in_channels: Number of input channels (default is 1 for grayscale images).
    - out_channels: Number of output channels, which determines the depth of the convolutional layer.
    - kernel_size: Size of the convolutional kernel (filter).
    - stride: Stride value for the convolution operation.
    - padding: Padding added to the input during convolution.
- The forward method defines the forward pass of the convolutional layer:
    - It applies the convolution operation to the input x.
    - ReLU activation function is applied to the convolutional features to introduce non-linearity.
- The output features represent the extracted features from the input image, ready to be passed to the next layer in the network.

Second Layer: Primary Capsules

Following the convolutional layer are the Primary Capsules. They take the feature maps and produce output vectors that represent the presence of features along with their properties like orientation and scale.

The code snippet for Primary Capsules:

```
class PrimaryCaps(nn.Module):
    def __init__(self, num_capsules=8, in_channels=256, out_channels=32):
        super(PrimaryCaps,self).__init__()
        self.capsules=nn.ModuleList([nn.Conv2d(in_channels, out_channels,
kernel_size=9, stride=2, padding=0) for _ in range(num_capsules)])
    def forward(self,x):
        batch_size=x.size(0)
        u=[capsule(x).view(batch_size,32 * 6 * 6,1) for capsule in self.c
apsules]
        u=torch.cat(u,dim=-1)
        u_squash=self.squash(u)
        return u_squash
```

Squashing

The squashing function is crucial for normalizing the output vectors of capsules. It ensures that the length of the output vector is a value between 0 and 1, which corresponds to the probability that a feature is present. The function is given by:

$$v_j = \frac{|| s_j^2 ||}{1 + || s_j^2 ||} s_j$$

where v_j is the output vector of capsule j and s_j is the total output.

The code snippet for squashing is as follows:

```
def squash(self,inp_tensor):
    sq_norm=(inp_tensor ** 2).sum(dim=-1,keepdim=True)
    scale=sq_norm/(1+sq_norm)
    output_tensor=scale * inp_tensor/torch.sqrt(sq_norm)
    return output_tensor
```

Module List

The final section would detail the creation of a module list in PyTorch using the nn.ModuleList container. This allows for a flexible and dynamic creation of the network layers, accommodating the variable number of capsules needed for different tasks.

Third Layer - Digit Capsules

This layer has 10 distinct capsules, each corresponding to a digit from 0 to 9. Each capsule will receive a 1152-dimensional vector from 8 primary capsules and outputs a 16-dimensional vector that captures not just the presence but the orientation and other important features of the digits.

Dynamic Routing is the key step here. Unlike traditional CNNs where data flow is static and unidirectional, capsules in Capsule Networks can dynamically route information. This means they can adjust the **coupling coefficients**, which are probabilities indicating the likelihood of information flowing from one capsule to another. This ability is crucial, especially when dealing with overlapping digits or digits appearing at various orientations.

The code snippet that illustrates dynamic routing:

```
def dynamic_routing(b_ij, u_hat, squash, routing_itrs=3):
    for itr in range(routing_itrs):
        c_ij = F.softmax(b_ij, dim=2)
        s_j = (c_ij * u_hat).sum(dim=2, keepdim=True)
        v_j = squash(s_j)
        if itr < routing_itrs-1:
            a_ij = (u_hat * v_j).sum(dim=-1, keepdim=True)
            b_ij = b_ij + a_ij
    return v_j
```

This function iteratively updates the coupling coefficients based on the **agreement** between predictions from lower-level capsules and higher-level capsules.

Routing by Agreement is the core part that will determine how the capsules transmit information. Capsules send output vectors to parent capsules in the next layer. The strength of these connections, the coupling coefficients, is not static. Instead it adjusts dynamically based on how well the prediction from a lower-level capsule agrees with the higher level capsule's current state. This agreement is measured by the scalar product between the predicted output vector of a lower capsule and the output vector of a higher capsule.

The information flow is controlled like below:

1. Each digit capsule starts with no prior knowledge about which lower-level capsule to trust more, hence the initial coupling coefficients are all equal.
2. **Predicted Output Vector (`u_hat`)**: This is a transformation of the lower-level capsules' outputs. These are predictions made by lower capsules about what the higher level capsules' output should be.
3. **Coupling Coefficients (`c_ij`)**: These are softmax outputs based on the log prior probabilities (`b_ij`) which represent the initial log odds that a lower level capsule's output should be coupled with a higher level capsule. These coefficients are updated based on the agreement between the predicted vectors (`u_hat`) and the actual output vectors of higher level capsules.
4. **Total Input (`s_j`)**: This is the weighted sum of all predicted output vectors (`u_hat`) from the lower level capsules, weighted by their updated coupling coefficients. This forms the total input to a capsule in the higher layer.
5. **Output Vectors (`v_j`)**: These are the outputs of each digit capsule obtained by applying a squashing function to the total input. The squashing function ensures that the length of the output vector is between 0 and 1 maintaining the direction but squashing the magnitude.

- The coupling coefficients are updated using the softmax function on the routing logits (b_{ij})

$$c_{ij} = \text{softmax}(b_{ij})$$

where c_{ij} represents the coupling coefficient indicating the likelihood of information flowing from one capsule to another.

- The total input (s_j) to each capsule is calculated as the weighted sum of the predicted outputs using the updated coupling coefficients:

$$s_j = \sum_i c_{ij} \cdot \hat{u}_{j|i}$$

where $\hat{u}_{j|i}$ represents the predicted output from the lower-level capsule.

- The output vectors (v_j) of capsules are obtained by applying the squash function to the total inputs:

$$v_j = \text{squash}(s_j)$$

The corresponding code snippet:

```
def dynamic_routing(b_ij, u_hat, squash, routing_itrs=3):
    for itr in range(routing_itrs):
        c_ij = F.softmax(b_ij, dim=2)  # Softmax applied to logits to for
m coupling coefficients
        s_j = (c_ij * u_hat).sum(dim=2, keepdim=True)  # Total input is t
he weighted sum of u_hat
        v_j = squash(s_j)  # Apply squashing function to normalize output
s

        # Update the logits based on agreement for the next iteration, ex
cept the last
        if itr < routing_itrs-1:
            a_ij = (u_hat * v_j).sum(dim=-1, keepdim=True)
            b_ij = b_ij + a_ij
    return v_j
```

This implementation of the DigitCaps layer has a network module that can transform inputs from the primary capsules into the outputs of the digit capsules.

```
class DigitCaps(nn.Module):
    def __init__(self, num_capsules=10, prev_layer_nodes=32*6*6, in_chann
els=8, out_channels=16):
    ...
      self.W = nn.Parameter(torch.randn(num_capsules, prev_layer_nodes, i
n_channels, out_channels))

    def forward(self, u):
        u = u[None, :, :, None, :]  # Expanding the dimensions of u for m
atrix multiplication
        ...
        u_hat = torch.matmul(u, W)  # Predicted output vectors by lower c
apsules
        b_ij = torch.zeros(*u_hat.size())  # Initializing the log priors
for routing
```

```
        ...
        v_j = dynamic_routing(b_ij, u_hat, self.squash, routing_itrs=3)
# Applying dynamic routing
        return v_j

    def squash(self, inp_tensor):  # Squashing function to normalize vect
or outputs
        ...
        output_tensor = scale * inp_tensor / torch.sqrt(sq_norm)
        return output_tensor
```

1. The `DigitCaps` module initializes a weight matrix `W`, which is used to transform input vectors from the primary capsules into predicted vectors (`u_hat`) for each possible digit.
2. **Forward Pass**: The method first expands the input tensor to match the dimensions required for matrix multiplication with the weight tensor `W`. It then computes the predicted output vectors (`u_hat`) and initializes the log prior probabilities (`b_ij`). Dynamic routing is applied to refine the coupling coefficients based on the agreement between these predictions and the actual outputs, iteratively updating the outputs (`v_j`).
3. **Squashing Function**: The `squash` function is crucial as it ensures that the output vectors have a magnitude between 0 and 1, which helps in interpreting the vector's length as the probability of the presence of a digit.

This structure of the DigitCaps layer will allow the network to effectively learn complex spatial hierarchies between different features, which is needed for accurate digit recognition. The integration of dynamic routing at this stage ensures that the network adapts to focus more on the most relevant features for the task at hand, demonstrating a significant advancement over traditional convolutional networks.

Decoder

The decoder is a crucial component of the Capsule Network, mainly to reconstructing the input image based on the output vectors from the DigitCaps layer. This process not only aids in the network's training by providing a form of self-supervision but also helps in verifying the network’s understanding of the input data.

Functionality of the Decoder

1. The decoder receives the 16-dimensional vectors from each of the 10 digit capsules. Each vector represents the properties and the instantiation parameters of a specific digit as captured by the network.

2. The decoder identifies the "correct" output vector, which is determined by the capsule with the largest vector magnitude. This magnitude acts as a confidence measure or a probability indicating the presence of the corresponding digit in the input image.
3. Using the identified correct vector, the decoder attempts to reconstruct the input image. This reconstructed image is compared against the actual input to measure the decoder's performance, typically using the Euclidean distance to quantify the difference or loss between the original and reconstructed images.

The implementation of the decoder has several steps, primarily focusing on transforming the capsule outputs back into the space of the original image dimensions.

```
class Decoder(nn.Module):
    def __init__(self, inp_vec_len=16, inp_capsules=10, hidden_dim=512):
        super(Decoder, self).__init__()
        input_dim = inp_vec_len * inp_capsules  # Calculating the total i
nput dimension
        self.linear_layers = nn.Sequential(  # Defining a sequence of lin
ear transformat
        ...
            nn.Sigmoid()  # Ensuring the output values are between 0 and
1
        )

    def forward(self, x):
        classes = (x ** 2).sum(dim=-1) ** 0.5  # Calculating the magnitud
e of each capsule's output vector
        classes = F.softmax(classes, dim=-1)  # Applying softmax to form
a probability distribution
        _, max_length_indices = classes.max(dim=1)  # Finding the index o
f the capsule with the maximum length
        sparse_matrix = torch.eye(10)  # Creating a sparse identity matri
x for selecting the correct vector
        ...
        reconstruction = self.linear_layers(flattened_x)  # Passing throu
gh the linear layers to reconstruct the image
        return reconstruction, y
```

- init: Sets up the linear transformations needed to process the 16-dimensional vectors into a 784-dimensional output (corresponding to a 28×28 pixel image), including intermediate activations to introduce non-linearities.

- **Forward Pass**: Calculates the magnitudes of the capsule outputs to determine which digit capsule has the highest confidence level. The output from this capsule is then used exclusively to reconstruct the original image. The use of a softmax function ensures that the reconstruction is based on the most probable digit representation as per the network's prediction.

The decoder plays a key role in both training the network via a reconstruction loss and providing a tangible output that can be compared against the original input, thus closing the loop on the learning process and allowing for continuous refinement of the network's capabilities.

Complete Capsule Network Architecture

1. **Convolutional Layers**: The model starts with convolutional layers, which are responsible for initial feature detection such as edges and textures. These layers operate similarly to traditional CNNs but serve primarily as feature extractors for the subsequent capsule layers.
2. **Primary Capsule Layer**: Following the convolutional layers, the primary capsule layer uses several capsule units to transform the feature maps from the convolutional layers into a set of initial capsule outputs. These outputs represent various features and parts of the input image in a way that preserves spatial hierarchies.
3. **Digit Capsule Layer**: The outputs from the primary capsules are then routed to the digit capsule layer. This layer consists of 10 distinct capsules, each corresponding to a digit (0-9). Here, dynamic routing takes place to determine which features contribute to the detection of each specific digit, refining the network's understanding and classification capabilities.
4. **Decoder**: The final component is the decoder. This module takes the output from the digit capsules and attempts to reconstruct the original input image. The quality and accuracy of this reconstruction serve as feedback to the network, helping it learn to encode only the most relevant features necessary for accurate digit recognition.

The complete Capsule Network model is implemented as a class in PyTorch, integrating all the components described above:

```
class CapsuleNetwork(nn.Module):
    def __init__(self):
        super(CapsuleNetwork, self).__init__()
        self.conv_layer = ConvLayer()  # Convolutional layer
        self.primary_capsules = PrimaryCaps()  # Primary capsule layer
        self.digit_capsules = DigitCaps()  # Digit capsule layer
        self.decoder = Decoder()  # Decoder

    def forward(self, imgs):
        primary_caps_output = self.primary_capsules(self.conv_layer(img
s))
```

```
        caps_output = self.digit_capsules(primary_caps_output).squeeze().
transpose(0,1)
        reconstr, y = self.decoder(caps_output)
        return caps_output, reconstr, y
```

1. **Initialization:** In the constructor of the `CapsuleNetwork` class, individual components of the model are initialized. These include the initial convolutional layer (`ConvLayer`), the primary capsules (`PrimaryCaps`), the digit capsules (`DigitCaps`), and the decoder (`Decoder`).
2. **Forward Pass:** The `forward` method defines the data flow through the network:
    - Input images first pass through the convolutional layer to produce feature maps.
    - These feature maps are then processed by the primary capsule layer, which outputs a set of initial capsule vectors.
    - These vectors are routed to the digit capsule layer, where dynamic routing occurs to finalize the capsule outputs.
    - The output of the digit capsule layer is then used by the decoder to reconstruct the input image, and to output the classification (`y`).

This structured approach allows the Capsule Network to leverage both hierarchical capsule processing and traditional convolutional features, ensuring detailed recognition and interpretation of complex image data. The model's ability to dynamically route information and reconstruct inputs makes it a powerful tool for tasks requiring nuanced understanding of visual data.

Margin Loss & Reconstruction Loss

Components of the Loss Function

1. **Margin Loss**: This component is designed to enhance the classification performance of the network. Margin loss ensures that the capsule output vectors for the correct class have a large magnitude, reflecting high confidence, while those for incorrect classes remain below a certain threshold. This differentiation helps in fine-tuning the capsule outputs for better discriminative learning.

Margin loss is used to ensure that the correct digit capsule produces a high output magnitude when the corresponding digit is present.

$$\text{Margin Loss} = \sum_k T_k \cdot \max(0, m^+ - ||v_k||)^2 + \lambda \cdot (1 - T_k) \cdot \max(0, ||v_k|| - m^-)^2$$

where:

- is the target output for capsule \(k \).

- v_k is the output vector of capsule \(k \).
- m^+ and m^- are the margin parameters for positive and negative classes, respectively.
- λ is a down-weighting factor for the loss of absent capsules.

2. **Reconstruction Loss**: The second component, reconstruction loss, focuses on the quality of the reconstructed images compared to the original inputs. By penalizing the network based on the Euclidean distance between the original and reconstructed images, this loss helps the network learn to preserve all necessary details, leading to a better understanding of the input data.

Reconstruction loss measures the dissimilarity between the original input images and their reconstructions produced by the decoder.

$$\text{Reconstruction Loss} = \text{MSE}(X, \hat{X})$$

where:

- X is the original input image.
- $\hat{X}$ is the reconstructed image.

Combined Loss Function

The combined loss function is designed to guide the network towards optimal performance by balancing classification accuracy with the fidelity of reconstructions. This balance ensures that the network does not overly focus on one aspect at the expense of another, making it robust and versatile for various image recognition tasks.

Detailed breakdown of the implementation of the custom loss function in the Capsule Network:

```
class CapsuleLoss(nn.Module):
    def __init__(self):
        super(CapsuleLoss, self).__init__()
        self.reconstruction_loss = nn.MSELoss(reduction='sum')  # Mean Sq
uared Error loss for reconstruction

    def forward(self, x, labels, imgs, reconstr):
        batch_size = x.size(0)
        v_c = torch.sqrt((x**2).sum(dim=2, keepdim=True))  # Calculate th
e magnitude of the capsule outputs
        left = F.relu(0.9 - v_c).view(batch_size, -1)  # Loss for correct
class
        right = F.relu(v_c - 0.1).view(batch_size, -1)  # Loss for incorr
ect classes
        margin_loss = labels * left + 0.5 * (1. - labels) * right
        margin_loss = margin_loss.sum()  # Summing up the margin loss acr
```

```
oss all instances

        imgs = imgs.view(reconstr.size()[0], -1)  # Reshaping images for
loss calculation
        reconstruction_loss = self.reconstruction_loss(reconstr, imgs)  #
Calculating the reconstruction loss

        return (margin_loss + 0.0005 * reconstruction_loss) / imgs.size
(0)  # Combining the losses
```

- **Initialization:** Sets up the MSE loss for reconstruction, configuring it to sum errors over all elements, which helps in emphasizing the importance of each pixel in the reconstruction process.
- **Forward Pass**: Computes both margin and reconstruction losses:
  - **Margin Loss**: Calculates for both correct and incorrect classifications. For correct classifications, the network is penalized if the capsule output's magnitude is below 0.9, and for incorrect classifications, if the magnitude exceeds 0.1.
  - **Reconstruction Loss**: Measures the mean squared error between the reconstructed and original images, ensuring the network learns to accurately reproduce the input.
- **Combined Loss**: The final loss is a weighted sum of the margin and reconstruction losses, normalized by the number of images in the batch. This normalization is crucial for maintaining loss magnitude consistency across different batch sizes.

By effectively integrating these two loss components, the Capsule Network is adeptly guided through a comprehensive learning process that sharpens both its classification acumen and its ability to reconstruct detailed images, thus enhancing its overall efficacy.

Training Process

Training a Capsule Network involves a sequence of steps designed to iteratively update the model based on the training data. These steps are crucial for effectively learning both the classification and the reconstruction tasks that define the Capsule Network's functionality.

Key Steps in the Training Process

1. **Clear Gradients**: Before each forward pass, it is essential to reset the gradients accumulated from the previous batch to prevent them from influencing the current batch's gradient calculations.

2. **Forward Pass**: The network processes the input data through all its layers, from the initial convolutional layers to the digit capsules and finally through the decoder, to produce both the classification outputs and the reconstructed images.
3. **Calculate Loss**: The combined loss (incorporating both margin and reconstruction losses) is computed to assess how well the model is performing with respect to both its primary and auxiliary tasks.
4. **Backward Pass**: Based on the computed loss, the network calculates the gradients for all trainable parameters. This step is crucial for learning, as it determines how each parameter should be adjusted to minimize the loss.
5. **Optimization Step**: Using the gradients calculated in the backward pass, the optimizer updates the model parameters. This step is critical for improving the network's performance over time.
6. **Record and Monitor Training Loss**: The loss for each batch is recorded, and periodically, the average loss over several batches is calculated and printed. This monitoring helps in assessing the model's learning progress and making necessary adjustments to the training process.

```
def train(capsule_net, criterion, optimizer, n_epochs, train_loader, prin
t_every=300):
    for epoch in range(1, n_epochs + 1):
        train_loss = 0.0
        capsule_net.train()  # Set the network to training mode
        for batch_i, (imgs, target) in enumerate(train_loader):
            optimizer.zero_grad()  # Clear gradients
            caps_output, reconstr, y = capsule_net(imgs)  # Forward pass
            loss = criterion(caps_output, target, imgs, reconstr)  # Calc
ulate loss
            loss.backward()  # Backward pass
            optimizer.step()  # Optimization step

            train_loss += loss.item()  # Accumulate the loss
            if batch_i % print_every == 0:  # Print average loss every fe
w batches
                print(f'Epoch: {epoch}/{n_epochs}, Batch: {batch_i}, Los
s: {train_loss/print_every}')
                train_loss = 0
```

Testing and Performance Evaluation

Testing the trained Capsule Network involves evaluating its performance using unseen test data. This step is critical to understanding how well the model generalizes beyond the data it was trained on.

1. **Evaluation on Test Data**: The network is tested using a separate dataset that was not involved in the training process. This helps in evaluating the model's effectiveness in real-world scenarios.
2. **Classification Accuracy and Loss Metrics**: The performance is measured in terms of classification accuracy and the combined loss. These metrics provide a quantitative measure of the model's predictive and reconstructive capabilities.
3. **Granular Performance Analysis**: The accuracy for each digit class is analyzed to identify any potential biases or weaknesses in the model's learning.
4. **Insights into Model Generalization**: By analyzing the test results, insights can be gained into how effectively the model can generalize its learning to new, unseen data.

```
def test(capsule_net, test_loader):
    capsule_net.eval()  # Set the network to evaluation mode
    test_loss, total_accuracy = 0, 0
    for imgs, target in test_loader:
        caps_output, reconstr, y = capsule_net(imgs)  # Forward pass
        loss = criterion(caps_output, target, imgs, reconstr)  # Calculat
e loss
        test_loss += loss.item()  # Accumulate loss
        accuracy = (y.argmax(dim=1) == target).float().mean()  # Calculat
e accuracy
        total_accuracy += accuracy.item()  # Accumulate accuracy
    print(f'Test Loss: {test_loss / len(test_loader)}, Accuracy: {total_a
ccuracy / len(test_loader)}')
```

This testing phase is essential for validating the robustness and reliability of the Capsule Network, ensuring that it performs well not just in a controlled training environment but also in practical applications where variability is greater.

Image Reconstructions

A critical aspect of assessing a Capsule Network's performance is evaluating how well it can reconstruct the original input images after processing them through its layers. This reconstruction is crucial for understanding the decoder’s effectiveness and the network’s ability to capture and represent essential data features.

Evaluating Decoder Performance

The decoder's ability to reconstruct images is an essential measure of the Capsule Network's effectiveness. The reconstructed images are compared to the original MNIST images to evaluate the fidelity of the reconstructions. This comparison helps highlight how well the network has learned to encode and decode the essential features of the images:

1. **Display of Original and Reconstructed Images**: By visualizing the original and reconstructed images side by side, one can directly assess the decoder's performance.
2. **Reconstruction Characteristics**: Typically, the reconstructed images should closely resemble the originals, capturing key features like shapes and edges. However, slight blurring or smoothing at the edges is common, indicating minor discrepancies in the network's ability to perfectly replicate every detail.

```
def show_images(original_imgs, reconstructed_imgs):
    num_images = 10  # Number of images to display
    fig, axes = plt.subplots(nrows=2, ncols=num_images, figsize=(20, 4))
# Create a figure with two rows of images
    for i in range(num_images):
        # Display original images
        ax = axes[0, i]
        ax.imshow(original_imgs[i].reshape(28, 28), cmap='gray')  # Resha
pe and display as grayscale
        ax.axis('off')  # Hide axes

        # Display reconstructed images
        ax = axes[1, i]
        ax.imshow(reconstructed_imgs[i].reshape(28, 28), cmap='gray')  #
Reshape and display as grayscale
        ax.axis('off')  # Hide axes

    plt.show()  # Display the figure

# Assuming 'imgs' are the original MNIST images and 'reconstr' are the ou
tputs from the decoder
show_images(imgs, reconstr)
```

- The function `show_images` takes two arrays of images: `original_imgs` and `reconstructed_imgs`. Each array contains a set of images to be displayed.
- **Plotting Images**: The function sets up a figure with two rows using `plt.subplots`. The top row displays the original images, and the bottom row displays the reconstructed images. Each

image is reshaped to its original dimensions (28×28 pixels) and displayed in grayscale.

- **Visualization**: Axes are turned off for a cleaner display, focusing solely on the image content. The `plt.show()` function renders the plot, allowing for visual inspection of the decoder's output.

This visualization technique provides a direct, intuitive means of evaluating the quality of the reconstructed images compared to the originals, highlighting the decoder's capabilities and areas where it might be lacking. Such evaluations are crucial for further tuning and improvements of the Capsule Network.

Task

Implement Capsule Networks on a New Dataset: You will apply the capsule network architecture to alternative datasets such as Fashion-MNIST or CIFAR-10. This exercise will enhance understanding of the model's adaptability across various image data types and allow exploration of its efficacy in more intricate settings.

References

https://arxiv.org/pdf/1710.09829.pdf

https://github.com/cezannec/capsule_net_pytorch/blob/master/Capsule_Network.ipynb

Beginner's Guide to Capsule Networks

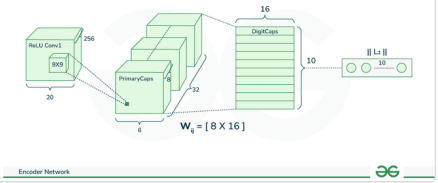
Explore and run machine learning code with Kaggle Notebooks | Using data from multiple data sources

https://www.kaggle.com/code/fizzbuzz/beginner-s-guide-to-capsule-networks

Introduction to Capsule Neural Networks | ML - GeeksforGeeks

A Computer Science portal for geeks. It contains well written, well thought and well explained computer science and programming articles, quizzes and practice/competitive programming/company interview Questions.

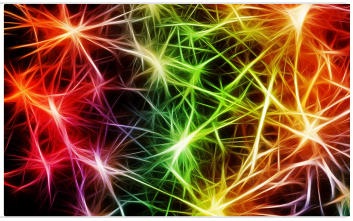
https://www.geeksforgeeks.org/capsule-neural-networks-ml/



Essentials of Deep Learning: Getting to know CapsuleNets (with Python codes)

Discover the concept behind capsule networks and compare the different neural network architecture in python. Learn about deep learning capsule networks.

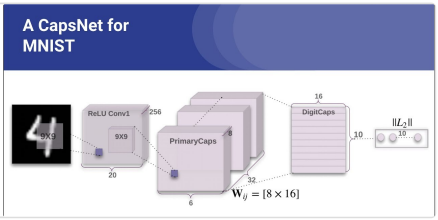
https://www.analyticsvidhya.com/blog/2018/04/essentials-of-deep-learning-getting-to-know-capsulenets/



Capsule Networks (CapsNets) – Tutorial

CapsNets are a hot new architecture for neural networks, invented by Geoffrey Hinton, one of the godfathers of deep learning.

https://www.youtube.com/watch?v=pPN8d0E3900



Introduction to Capsule Networks | Paperspace Blog

This post covers an introduction to Capsule Networks, a network architecture which mimics human biology to better model hierarchical relationships.

https://blog.paperspace.com/capsule-networks/