Final Exam Information

Final Exam

- Contributes 40% to the total mark
- Hurdle to pass the unit:
  - At least 45% in in-semester assessments (2 quizzes + 2 assignments)
  - AND, at least 45% in final exam
  - AND, at least 50% overall mark
- Format: 10 mins reading + *2 hours*
- *Closed* book
  - CALCULATOR is allowed, highly recommended
  - TWO pages of notes (one A4 sheet)

Exam Information

Authorised Materials

- Closed Book with Specifically permitted items
- Physical / Virtual Calculator
- Working Sheets: Yes
- Answer Sheets: 10
  What's an answer sheet
- Notes - Double sided A4 x 1 - Physical Only

It contains THREE (3) parts with a total of **100 marks**:

- **Part A** contains **12** multiple-choice questions, together they are worth a total of **25 marks**. For questions with more than one answer, a correct choice will receive a partial mark and incorrect choices will reduce the mark. To receive full marks, only correct choices must be selected.
- **Part B** contains **8** short workout questions, worth **40 marks**. These questions typically require short knowledge answers and calculations based on the knowledge you have learned from the unit. Having the calculator handy for these questions is recommended.
- **Part C** contains **10** mixed and written-answer questions, worth **35 marks**. These questions typically assess the knowledge and understanding of lecture contents.
- Once the exam duration is finished, your exam will automatically submit. Please ensure you finalise your answers before the end of the allocated exam time.

- Exam difficulty
  - 80%: easy and medium questions (directly from lectures, tutorials, and quizzes)
  - 10%: quite challenging questions (deducting the answers from lectures, tutorials, and quizzes)
  - 10%: challenging questions (connecting the dots and high level understanding knowledge)

Exam

- Contents:
  - Part A: multiple-choice questions
  - Part B: short workout questions
  - Part C: knowledge and written answer questions
- What might be tested?
  - Lecture contents from week 01 to week 11 (refer to the revision slides)
  - Tutorial contents (code reading) and assignments
  - Will focus on the understanding of subjects, concepts and mechanism (e.g., how attention work, etc)
- What will NOT be tested?
  - Appendices from the lectures
  - Memorization of existing network specifications such as AlexNet, VGGNet, ResNet, Transformer, BERT, and etc. Moreover, when questions asked are relevant to these architectures, the relevant codes will be provided.

Exam

- Tips:
  - Revising questions from assignment 1 and two quizzes
  - **No coding question**, read the code snippet and answer the questions (should be straightforward if you've done the tutorials)
  - Understand essential model expressions, i.e., mathematical expressions for variables in FNN, CNN, RNN, LSTM, etc
  - Materials on appendices won't be tested, revise the lectures

Part A: multiple-choice questions

- Similar to the quizzes, difficulty level $\approx \frac{1}{2}(\text{Test 1} + \text{Test 2})$

Question 7

Given a sequence of random variables $x_1, x_2, \ldots, x_n$, which of the following equation expresses the product rule in probability theory?

1 Mark

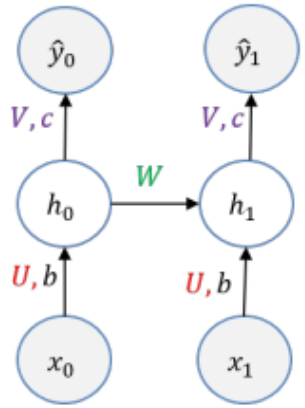
Select one:

- A. $p(x_1, x_2, \ldots, x_n) = 1.$
- B. $p(x_1, x_2, \ldots, x_n) = p(x_1)p(x_2)\ldots p(x_n).$
- C. $p(x_1, x_2, \ldots, x_n) = p(x_n|x_{n-1})p(x_{n-1}|x_{n-2})\ldots p(x_2|x_1)p(x_1).$
- D. $p(x_1, x_2, \ldots, x_n) = p(x_n|x_{1:n-1})p(x_{n-1}|x_{1:n-2})\ldots p(x_2|x_1)p(x_1).$

Part B: workout questions

- Calculation based on specific architecture knowledge
  - Question: *What is the result of 2d convolution using a filter xxx on an image yyy with zero padding size = 1, stride = 2 in all directions?*

Consider the first two time-slices of a RNN architecture for the regression task with weight matrices and bias terms shown in the following figure. Instead of using tanh() as in a standard RNN, assuming that the activation ReLU() is used in all cases. Answer the following questions:



$$V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, c = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$x_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad x_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

Question 14

Without using the any specific values on the right-hand-side of the figure, write down the analytical expressions for h_0 and h_1. (2 Marks)

Please answer question on your blank piece of paper.

- After your exam finishes, you'll have extra time to access your phone to scan a QR code and upload your answer.
- Clearly label each page with Student ID and this question number (and sub part if applicable) (for example, 'Question 7a')
- Do not write your Name on it

No. of answer sheets: 1

Question 15

Now given the values for x_0, x_1, b, c and matrices U, V, W as shown on the right-hand-side of the figure, calculate h_0, h_1 and then the output $\hat{y}_1$. (3 Marks)

Please answer question on your blank piece of paper.

- After your exam finishes, you'll have extra time to access your phone to scan a QR code and upload your answer.
- Clearly label each page with Student ID and this question number (and sub part if applicable) (for example, 'Question 7a')
- Do not write your Name on it

No. of answer sheets: 1

Part C: short-answer questions

- Assessing mid-high level knowledge
  - *Question: given the following architecture, write down the mathematical expression for xxx , its advantages compared with Y, etc.*

The statement below applies to Q.23 - Q.24

Consider a Convolution Neural Network (CNN) with the model parameter θ. Specifically, given an image x with the ground-truth label y, the CNN returns the prediction probabilities f(x;θ) over M classes and suffers the loss l(f(x;θ),y) where l is a loss function (e.g., the cross-entropy loss).

a) Adversarial examples are a serious issue of CNNs. Give a definition of adversarial examples. Give a practical example to explain why adversarial examples circumvent the applications of CNNs in reality. **(3 points)**

b) Given a benign example x and the ϵ-ball $B_\epsilon=\{x':\|x'-x\|_\infty\leq\epsilon\}$, describe and give the formula to find out a targeted adversarial example for x. **(3 points)**

c) Given a benign example x and the ϵ-ball $B_\epsilon=\{x':\|x'-x\|_\infty\leq\epsilon\}$, describe and give the formula to find out a untargeted adversarial example for x. **(3 points)**

Please answer the question on your blank piece of paper.

- After your exam finishes, you'll have **extra time** to access your phone to scan a **QR code** and **upload** your answer.
- Clearly label each page with **Student ID** and **this question number** (and **sub part** if applicable) (for example, 'Question 7a')
- **Do not** write your Name on it

No. of answer sheets: 2

a) Given a mini-batch $B=\{(x_1,y_1),\ldots,(x_b,y_b)\}$ at an iteration, describe how to perform adversarial training for this mini-batch to improve model robustness. **(3 points)**

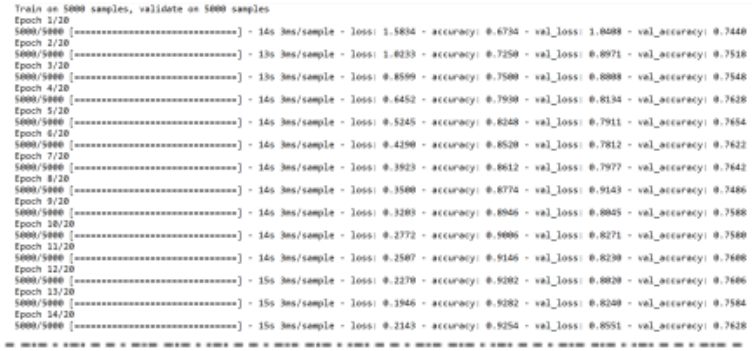
b) How adversarial training is similar to data augmentation? How adversarial training is different from data augmentation? **(3 points)**

Please answer the question on your blank piece of paper.

- After your exam finishes, you'll have **extra time** to access your phone to scan a **QR code** and **upload** your answer.
- Clearly label each page with **Student ID** and **this question number** (and **sub part** if applicable) (for example, 'Question 7a')
- **Do not** write your Name on it

6 Marks

By observing the training outcome from a CNN architecture as shown in the figure below, answer the following questions:



30a)

2 Marks

(a) Identify the issue of the training outcome with explanation. (2 marks)

Report question issue

30b)

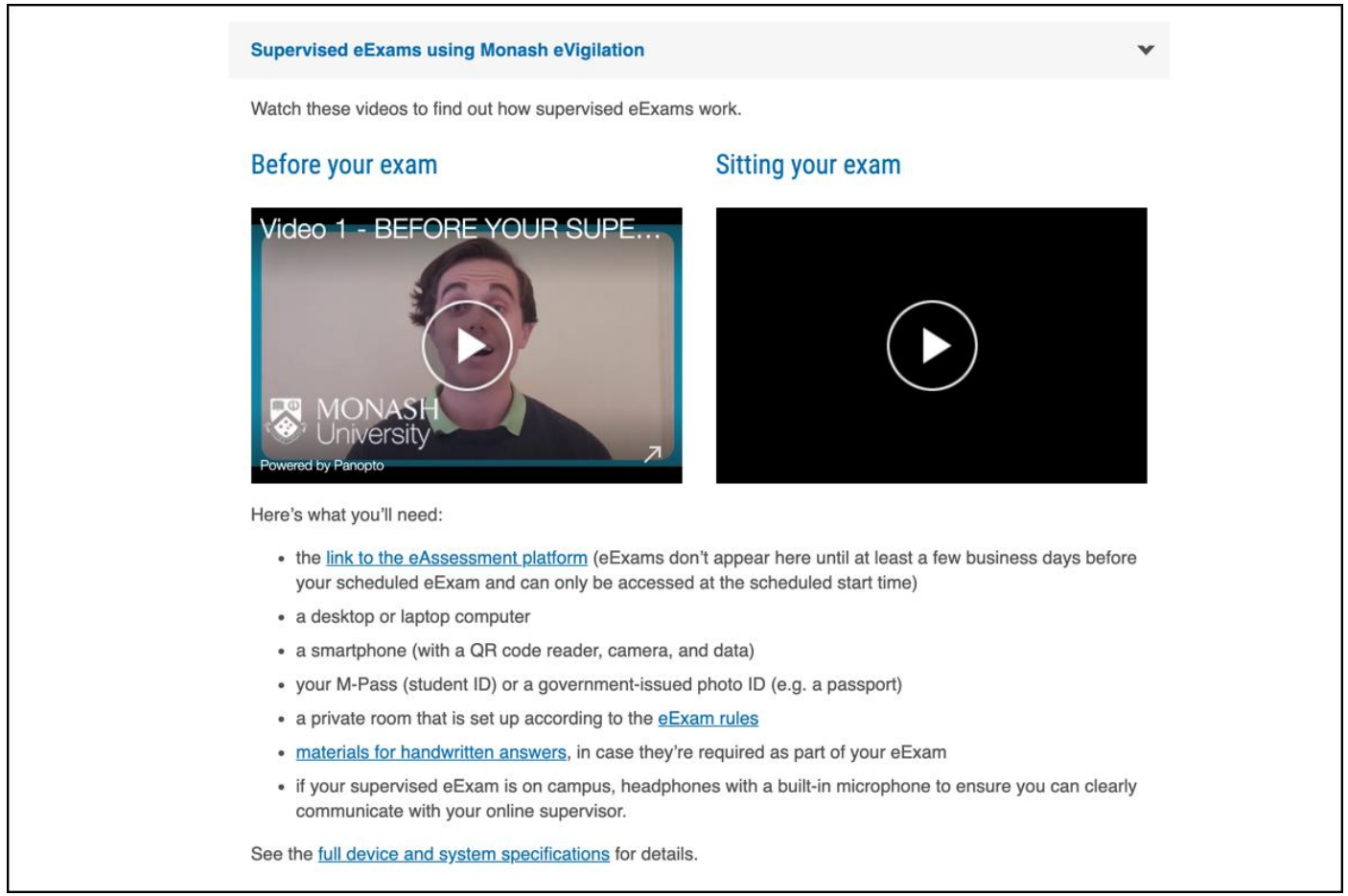
4 Marks

(b) Suggest and describe TWO (2) methods that can be used to solve the above issue.

eExam

eExam rules: https://www.monash.edu/exams/electronic-exams/rules

Need help during exam: https://www.monash.edu/exams/electronic-exams/help



Supervised eExams using Monash eVigilation

Watch these videos to find out how supervised eExams work.

Before your exam

Sitting your exam

Here's what you'll need:

- the link to the eAssessment platform (eExams don't appear here until at least a few business days before your scheduled eExam and can only be accessed at the scheduled start time)
- a desktop or laptop computer
- a smartphone (with a QR code reader, camera, and data)
- your M-Pass (student ID) or a government-issued photo ID (e.g. a passport)
- a private room that is set up according to the eExam rules
- materials for handwritten answers, in case they're required as part of your eExam
- if your supervised eExam is on campus, headphones with a built-in microphone to ensure you can clearly communicate with your online supervisor.

See the full device and system specifications for details.

Hybrid question

22 of 28

Please answer question on your blank piece of paper.

- After your exam finishes, you'll have **extra time** to access your phone to scan a **QR code** and **upload** your answer.
- Clearly label each page with **Student ID** and **this question number** (and **sub part** if applicable) (for example, 'Question 7a')
- **Do not** write your Name on it

No. of answer sheets: 2

☐ I have answered this question

Report question issue Notes

Question 22 Notes

☐ Unsure

Reading and Explaining Codes

- For CNNs and vision data
  - Conv2d, Conv2dTranspose, MaxPool2d, AdaptiveAvg2d,
  - Flatten, Unflatten, BatchNorm2d, BatchNorm1d, Linear
- For RNNs and sequential data
  - nn.Embedding, nn.GRU, nn.LSTM, nn.RNN

nn.Conv2D

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size,
                stride=1, padding=0,
                dilation=1, groups=1, bias=True, padding_mode='zeros')
```

- What it does:
  - nn.Conv2d performs a 2D convolution operation, essential for extracting features from images in CNNs. Imagine a filter sliding across your image, detecting patterns like edges, corners, and textures.
- Output shape
  - $output\_size = \left\lfloor \frac{input\_size + 2 * padding - kernel\_size}{stride} \right\rfloor + 1$

Key parameters:

- `in_channels` : Number of channels in the input image (e.g., 3 for RGB).
- `out_channels` : Number of filters to apply (controls the number of output features).
- `kernel_size` : Size of the filter (e.g., 3x3, 5x5).
- `stride` : Step size for sliding the filter (larger stride = more downsampling).
- `padding` : Adds zeros around the image borders (helps control output size).

nn.Conv2dTranspose

```
torch.nn.ConvTranspose2d(in_channels, out_channels, kernel_size,
                         stride=1, padding=0,
                         output_padding=0, groups=1,
                         bias=True, dilation=1, padding_mode='zeros')
```

- What it does:
  - nn.ConvTranspose2d performs a transposed convolution, often called "deconvolution" (though not technically accurate).
  - It's used to upsample feature maps, effectively increasing their spatial dimensions. This is crucial in tasks like image generation for example in generators of DCGAN.
  - Think of it as the reverse of nn.Conv2d: While nn.Conv2d extracts features and downsamples, nn.ConvTranspose2d takes those features and reconstructs a higher-resolution output.
- Output size
  - $output\_size = (input\_size - 1) * strides - 2 * padding + kernel\_size$
  - You can obtain the above formula by reversing the Conv2d formula: $output\_size = \left\lfloor \frac{input\_size + 2 * padding - kernel\_size}{stride} \right\rfloor + 1$

Key parameters:

- `in_channels` : Number of channels in the input feature map.
- `out_channels` : Number of output channels.
- `kernel_size` : Size of the transposed convolution kernel.
- `stride` : Controls the upsampling factor (larger stride = larger output).
- `padding` : Adds zeros around the input (influences output size).
- `output_padding` : Adds extra zeros to one side of the output (for precise size control).

nn.Linear, nn.Flatten, nn.Unflatten

```
torch.nn.Linear(in_features, out_features,
                bias=True, device=None, dtype=None)
```

- What it does:
  - nn.Linear implements a fully connected layer, also known as a linear transformation. It connects every neuron in one layer to every neuron in the next layer. This is a fundamental building block for most neural networks.
  - Think of it as a matrix multiplication: It takes an input vector and multiplies it by a weight matrix, then adds a bias vector. This allows the network to learn complex relationships between features.

Key parameters:

- `in_features`: The number of input features (size of the input vector).
- `out_features`: The number of output features (size of the output vector).
- `bias`: If `True`, adds a learnable bias to the output (usually helpful).

```
torch.nn.Flatten(start_dim=1, end_dim=-1)
```

- `start_dim`: The first dimension to flatten (inclusive). Default is 1, which means it starts flattening from the second dimension.
- `end_dim`: The last dimension to flatten (inclusive). Default is -1, which means it flattens until the last dimension.

Why is the default `start_dim=1`?

In many deep learning scenarios, the first dimension of a tensor represents the batch size. By default, `nn.Flatten` keeps the batch dimension separate, which is often what you want when processing batches of data.

```
torch.nn.Unflatten(dim, unflattened_size)
```

- `dim`: The dimension to unflatten. This is the dimension that will be split into the new shape specified by `unflattened_size`.
- `unflattened_size`: A tuple or a list specifying the new shape of the unflattened dimension.

Let's say you have a tensor `x` with shape `[128, 7*7*128]`. You want to unflatten the second dimension (`dim=1`) back into a shape of `[128, 7, 7]`. You would use:

Python

```
unflatten = nn.Unflatten(dim=1, unflattened_size=(128, 7, 7))
output = unflatten(x)  # output shape will be [128, 128, 7, 7]
```

nn.Flatten and nn.AdaptiveAvg2d

```
torch.nn.Flatten(start_dim=1, end_dim=-1)
```

- `start_dim`: The first dimension to flatten (inclusive). Default is 1, which means it starts flattening from the second dimension.
- `end_dim`: The last dimension to flatten (inclusive). Default is -1, which means it flattens until the last dimension.

Why is the default `start_dim=1`?

In many deep learning scenarios, the first dimension of a tensor represents the batch size. By default, `nn.Flatten` keeps the batch dimension separate, which is often what you want when processing batches of data.

```
# Input tensor
x = torch.randn(64, 128, 7, 7)

# Flatten layer
flatten_layer = nn.Flatten()  # Default: start_dim=1, en
output_flatten = flatten_layer(x)
print("Output shape of Flatten:", output_flatten.shape)
```

- **Flatten Layer:**
  - `nn.Flatten()` with default arguments keeps the batch dimension (dim=0) and flattens the remaining dimensions (dim=1, 2, 3).
  - This effectively collapses the spatial dimensions (7x7) and the channel dimension (128) into a single feature dimension of size 128 * 7 * 7 = 6272.
  - The output shape becomes `[64, 6272]`.

```
torch.nn.AdaptiveAvgPool2d(output_size)
```

- `output_size`: This can be:
  - A single integer to specify the same output size for both height and width.
  - A tuple of two integers `(height, width)` to specify different output sizes.

What does `nn.AdaptiveAvgPool2d` do?

It performs adaptive average pooling on a 2D input (like an image). Unlike regular average pooling, where you specify the kernel size and stride, adaptive pooling lets you directly specify the desired output size. The pooling kernel size is automatically calculated to achieve the desired output size.

```
# Input tensor
x = torch.randn(64, 128, 7, 7)

# AdaptiveAvgPool2d layer with output_size = (1, 1)
adaptive_avg_pool = nn.AdaptiveAvgPool2d((1, 1))

# Apply adaptive average pooling to the input
# Output shape: torch.Size([64, 128, 1, 1])
output = adaptive_avg_pool(x)
```

Input: We start with the same random input tensor `x` with shape `[64, 128, 7, 7]`. Think of it as 64 images, each with 128 feature maps of size 7x7.

AdaptiveAvgPool2d: This time, we create an `AdaptiveAvgPool2d` layer with `output_size=(1, 1)`. Instead of shrinking the 7x7 feature maps to 3x3, we want to shrink them all the way down to a single pixel (1x1).

Output: When we apply this layer (`adaptive_avg_pool(x)`), it takes each 7x7 feature map and calculates the average of all the values within that map. This average value becomes the single pixel in the 1x1 output feature map. We still have 64 images, and each image still has 128 feature maps, but now each feature map is just a single pixel representing the average of the original 7x7 map. The final output shape is `[64, 128, 1, 1]`.

BatchNorm2d and MaxPool2D

```
torch.nn.MaxPool2d(kernel_size, stride=None, padding=0,
                   dilation=1, return_indices=False, ceil_mode=False)
```

What it does:

`nn.MaxPool2d` performs max pooling, a downsampling operation that reduces the spatial dimensions (height and width) of an input image or feature map. It works by dividing the input into non-overlapping rectangular regions (pooling windows) and outputting the maximum value within each region.

Think of it as a feature magnifier:

Max pooling highlights the most prominent features in each region, making the network more robust to small variations and noise in the input.

Key parameters:

- `kernel_size`: The size of the pooling window (e.g., 2x2, 3x3).
- `stride`: The step size for sliding the pooling window (larger stride = more downsampling).
- `padding`: Adds zeros around the input borders (influences output size).

```
torch.nn.BatchNorm2d(num_features,
                     eps=1e-05, momentum=0.1,
                     affine=True, track_running_stats=True)
```

What it does:

`nn.BatchNorm2d` applies batch normalization to 2D inputs like images. It normalizes the activations within a batch by subtracting the batch mean and dividing by the batch standard deviation. This helps stabilize and speed up the training of neural networks.

Think of it as a data stabilizer:

It prevents internal covariate shift, where the distribution of activations changes during training. This makes learning more efficient and less sensitive to initialization.

Key parameters:

- `num_features`: The number of channels in the input feature map.
- `eps`: A small value added to the denominator for numerical stability.
- `momentum`: The momentum for the running mean and variance.
- `affine`: If `True`, uses learnable affine parameters (scale and shift).
- `track_running_stats`: If `True`, tracks the running mean and variance.

Example 1 of CNN

- If we input **random_input** [128, 3, 28, 28] to the CNN, what are the shapes of x,h1,h2,h3,h4,h5?

```python
import torch.nn as nn
import torch

class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, stride=1, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, stride = 2,kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 64, stride =2, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(64)  # BatchNorm for the third layer
        self.flatten = nn.Flatten()
        self.fc = nn.Linear(64 * 7 * 7, 10)  # Adjusted input features for t

    def forward(self, x):
        h1 = self.conv1(x)
        h1 = self.bn1(h1)
        h1 = torch.relu(h1)
        h2 = self.conv2(h1)
        h2 = self.bn2(h2)
        h2 = torch.relu(h2)
        h3 = self.conv3(h2)  # Pass through the third convolutional layer
        h3 = self.bn3(h3)  # Apply batch normalization
        h3 = torch.relu(h3)
        h4 = self.flatten(h3)
        h5 = self.fc(h4)
        return h5
```

```python
# Create an instance of the CNN
model = SimpleCNN()

# Generate a random input tensor with shape [128, 3, 28, 28]
# (batch size 128, 3 channels, 28x28 image size)
random_input = torch.randn(128, 3, 28, 28)

# Feed the input to the CNN
output = model(random_input)
```

Example 2 of CNN

- If we input **random_input** [128, 3, 28, 28] to the CNN, what are the shapes of x,h1,h2,h3,h4,h5?

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, stride=1, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, stride = 2,kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 64, stride =2, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(64)  # BatchNorm for the third layer
        self.gap = nn.AdaptiveAvgPool2d((1, 1))
        self.flatten = nn.Flatten()
        self.fc = nn.Linear(64, 10)  # Adjusted input features for the FC la

    def forward(self, x):
        h1 = self.conv1(x)
        h1 = self.bn1(h1)
        h1 = torch.relu(h1)
        h2 = self.conv2(h1)
        h2 = self.bn2(h2)
        h2 = torch.relu(h2)
        h3 = self.conv3(h2)  # Pass through the third convolutional layer
        h3 = self.bn3(h3)  # Apply batch normalization
        h3 = torch.relu(h3)
        h4 = self.gap(h3)
        h4 = self.flatten(h4)
        h5 = self.fc(h4)
        return h5
```

```
# Create an instance of the CNN
model = SimpleCNN()

# Generate a random input tensor with shape [128, 3, 28, 28]
# (batch size 128, 3 channels, 28x28 image size)
random_input = torch.randn(128, 3, 28, 28)

# Feed the input to the CNN
output = model(random_input)
```

Example of Conv2dTranspose

- What is the shape of the output?

```python
import torch
import torch.nn as nn

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(100, 7 * 7 * 256, bias=False),  # Dense layer in Keras is equivalent to Linear in PyTorch
            nn.BatchNorm1d(7 * 7 * 256),
            nn.LeakyReLU(),
            nn.Unflatten(1, (256, 7, 7)),  # Reshape in Keras is equivalent to Unflatten in PyTorch

            nn.ConvTranspose2d(256, 128, kernel_size=(5, 5), stride=(3, 3), padding=(2, 2), output_padding=(0, 0), bias=False)
        )

    def forward(self, x):
        return self.model(x)

# Example usage
model = Model()
x = torch.randn(16, 100)  # Example input with batch size 16 and input shape (100,)
output = model(x)
print(output.shape)  # Should match the output shape after the transpose convolution
```

nn.Embedding, nn.GRU, nn.LSTM, nn.RNN

```
nn.Embedding(num_embeddings, embedding_dim, padding_idx=None,
             max_norm=None, norm_type=2.0, scale_grad_by_freq=False,
             sparse=False, _weight=None)
```

What it does:

`nn.Embedding` creates a lookup table that maps discrete indices (like word IDs) to dense vector representations (embeddings). This is crucial for working with categorical data in neural networks, especially in natural language processing (NLP).

Think of it as a dictionary for your model:

It allows your network to understand and process words, categories, or any discrete entities by representing them as meaningful vectors.

Key parameters:

- `num_embeddings` : The size of the vocabulary (total number of unique words or categories).
- `embedding_dim` : The size of each embedding vector (the dimensionality of the vector space).
- `padding_idx` : If specified, the entries at `padding_idx` do not contribute to the gradient; therefore, the embedding vector at `padding_idx` is not updated during training, i.e. it remains as a fixed "pad".

```
nn.LSTM(input_size, hidden_size, num_layers,
        bias=True, batch_first=False, dropout=0,
        bidirectional=False, proj_size=0)
```

What it does:

`nn.LSTM` implements a Long Short-Term Memory network, a type of recurrent neural network (RNN) that excels at processing sequential data like text, time series, and speech. It can learn long-term dependencies and patterns in sequences, overcoming the limitations of traditional RNNs.

Think of it as a memory cell:

LSTMs have a special memory cell that can store information for long periods. They use gates (input, forget, output) to control the flow of information, allowing them to selectively remember or forget past inputs.

Key parameters:

- `input_size` : The number of features in the input.
- `hidden_size` : The number of features in the hidden state.
- `num_layers` : The number of LSTM layers stacked on top of each other.
- `bias` : If `False` , then the layer does not use bias weights `b_ih` and `b_hh` . Default: `True`
- `batch_first` : If `True` , then the input and output tensors are provided as `(batch, seq, feature)` . Default: `False`
- `dropout` : If non-zero, introduces a Dropout layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `dropout` . Default: 0

nn.Embedding, nn.GRU, nn.LSTM, nn.RNN

```
nn.GRU(input_size, hidden_size, num_layers,
       bias=True, batch_first=False, dropout=0,
       bidirectional=False)
```

What it does:

`nn.GRU` implements a Gated Recurrent Unit, another type of recurrent neural network (RNN) designed for sequential data. It's similar to LSTM but with a simpler architecture, making it computationally more efficient.

Think of it as a streamlined memory cell:

GRUs have a gating mechanism to control the flow of information, allowing them to learn and remember patterns in sequences. They combine the forget and input gates of an LSTM into a single "update gate," making them less complex.

Key parameters:

- `input_size`: The number of features in the input.
- `hidden_size`: The number of features in the hidden state.
- `num_layers`: The number of GRU layers stacked on top of each other.
- `bias`: If `False`, then the layer does not use bias weights `b_ih` and `b_hh`. Default: `True`
- `batch_first`: If `True`, then the input and output tensors are provided as `(batch, seq, feature)`. Default: `False`
- `dropout`: If non-zero, introduces a Dropout layer on the outputs of each GRU layer except the last layer, with dropout probability equal to `dropout`. Default: 0

```
nn.RNN(input_size, hidden_size, num_layers,
       nonlinearity='tanh', bias=True, batch_first=False, dropout=0,
       bidirectional=False)
```

What it does:

`nn.RNN` provides a basic implementation of a recurrent neural network. It processes sequential data by iterating through the input sequence and maintaining a hidden state that captures information from previous time steps.

Think of it as a loop with memory:

RNNs apply the same weights to each element in the sequence, but the output depends not only on the current input but also on the previous inputs through the hidden state. This allows them to learn temporal dependencies in data.

Key parameters:

- `input_size`: The number of features in the input.
- `hidden_size`: The number of features in the hidden state.
- `num_layers`: The number of recurrent layers stacked on top of each other.
- `nonlinearity`: The non-linearity to use. Can be either 'tanh' or 'relu'. Default: 'tanh'
- `bias`: If `False`, then the layer does not use bias weights `b_ih` and `b_hh`. Default: `True`
- `batch_first`: If `True`, then the input and output tensors are provided as `(batch, seq, feature)`. Default: `False`
- `dropout`: If non-zero, introduces a Dropout layer on the outputs of each RNN layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- `bidirectional`: If `True`, becomes a bidirectional RNN. Default: `False`

Example of LSTM

```python
class RNNModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim):
        super(RNNModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm1 = nn.LSTM(embedding_dim, 128, batch_first=True)
        self.lstm2 = nn.LSTM(128, 256, batch_first=True)
        self.lstm3 = nn.LSTM(256, 128, batch_first=True)
        self.fc = nn.Linear(128, 10)

    def forward(self, x):
        x = self.embedding(x)  # Embed the input tokens
        print(x.shape)
        # Pass through the LSTM layers
        h1, _ = self.lstm1(x)  # Ignore the hidden state output
        print(h1.shape)
        h2, _ = self.lstm2(h1)
        print(h2.shape)
        h3, _ = self.lstm3(h2)
        print(h3.shape)
        out = self.fc(h3)  # Pass through the fully connected layer
        return out
```

```python
vocab_size = 10000  # Example vocabulary size
embedding_dim = 100

model = RNNModel(vocab_size, embedding_dim)

# Generate random input
#batch size = 64 and seq_len = 40
input_seq = torch.randint(0, vocab_size, (64, 40))

# Forward pass
output = model(input_seq)

print("Output shape:", output.shape)  # Output sha
```

```
torch.Size([64, 40, 100])
torch.Size([64, 40, 128])
torch.Size([64, 40, 256])
torch.Size([64, 40, 128])
```

```python
class RNNModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim):
        super(RNNModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm1 = nn.LSTM(embedding_dim, 128, batch_first=True)
        self.lstm2 = nn.LSTM(128, 256, batch_first=True)
        self.lstm3 = nn.LSTM(256, 128, batch_first=True)
        self.fc = nn.Linear(128, 10)

    def forward(self, x):
        x = self.embedding(x)  # Embed the input tokens
        print(x.shape)
        # Pass through the LSTM layers
        h1, _ = self.lstm1(x)  # Ignore the hidden state output
        print(h1.shape)
        h2, _ = self.lstm2(h1)
        print(h2.shape)
        h3, _ = self.lstm3(h2)
        h3 = h3[:, -1, :]  # Take the last time step output
        print(h3.shape)
        out = self.fc(h3)  # Pass through the fully connected layer
        return out
```

```python
vocab_size = 10000  # Example vocabulary size
embedding_dim = 100

model = RNNModel(vocab_size, embedding_dim)

# Generate random input
#batch size = 64 and seq_len = 40
input_seq = torch.randint(0, vocab_size, (64, 40))

# Forward pass
output = model(input_seq)
```

```
torch.Size([64, 40, 100])
torch.Size([64, 40, 128])
torch.Size([64, 40, 256])
torch.Size([64, 128])
```

Example of GRU

```python
class RNNModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim):
        super(RNNModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.gru1 = nn.GRU(embedding_dim, 128, batch_first=True)
        self.gru2 = nn.GRU(128, 256, batch_first=True)
        self.gru3 = nn.GRU(256, 128, batch_first=True)
        self.fc = nn.Linear(128, 10)

    def forward(self, x):
        x = self.embedding(x)  # Embed the input tokens
        print(x.shape)
        # Pass through the GRU layers
        h1, _ = self.gru1(x)  # Ignore the hidden state output
        print(h1.shape)
        h2, _ = self.gru2(h1)
        print(h2.shape)
        h3, _ = self.gru3(h2)
        h3 = h3[:, -1, :]  # Take the last time step output
        print(h3.shape)
        out = self.fc(h3)  # Pass through the fully connected layer
        return out
```

```python
vocab_size = 10000  # Example vocabulary size
embedding_dim = 100

model = RNNModel(vocab_size, embedding_dim)

# Generate random input
#batch size = 64 and seq_len = 40
input_seq = torch.randint(0, vocab_size, (64, 40))

# Forward pass
output = model(input_seq)
```

```
torch.Size([64, 40, 100])
torch.Size([64, 40, 128])
torch.Size([64, 40, 256])
torch.Size([64, 128])
```

Thank you and acknowledgement!

Student Evaluation of Teaching and Units (SETU)

At Monash University, we are always seeking ways to improve your learning experience. One way we do this is by asking for your feedback on the content, structure, assessment tasks and learning technologies used in this unit. Please take the time to complete this survey and help us to improve your learning experience.

Please give us feedback via SETU!

Thanks for your attention!

Question time