Assignment 2

Luis Vitor Zerkowski - 14895730

December 1, 2023

1 Transfer Learning for CNNs

Question 1.1

1.1.a

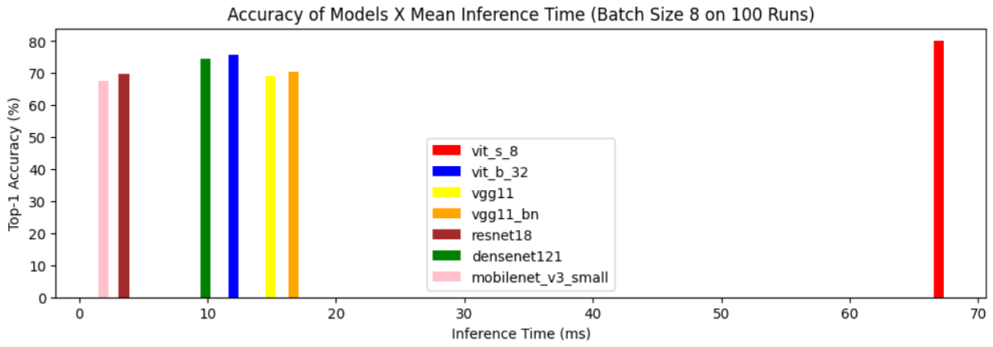

Figure 1: Graph of Top-1 accuracy vs mean inference time for each model with a batch size of 8 on 100 runs.

Figure 1 shows a correlation between Top-1 and mean inference time: the better the model, the slower it is. `VGG11` models, with and without batch normalization, break this pattern a bit comparing to the other models, but still respect it between each other.

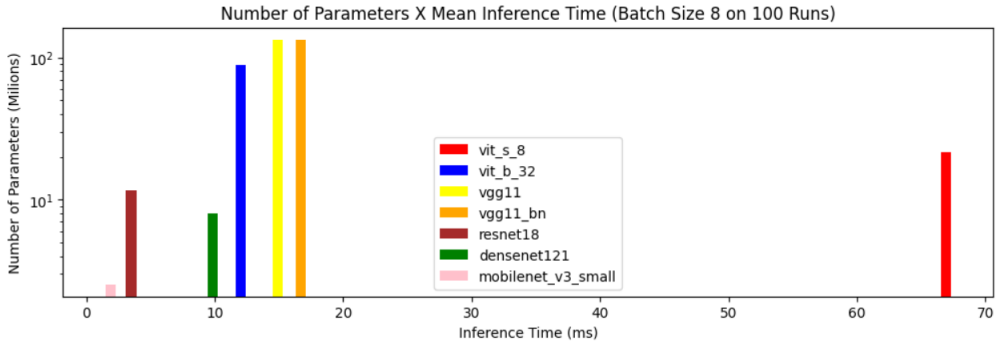

Figure 2: Graph of number of parameters vs mean inference time for each model with a batch size of 8 on 100 runs.

Figure 2 shows that the inference speed doesn't scale proportionately to the number of parameters. `ViT-S/8` model, for example, has a much larger mean inference time than `VGG11`, but almost five times less parameters. An analogous pattern breaking phenomenon happens to `ResNet18` and `DenseNet121`.

Model	Mean inference time for one frame (ms)
VGG11	3.80
VGG11-bn	3.92
ResNet18	**1.04**
DenseNet121	6.01
MobileNet-v3-Small	1.89
ViT-B/32	2.57
ViT-S/8	10.36

Table 1: Table with model vs mean inference time for one frame on 100 runs.

1.1.b

Without `torch.no_grad()` the inference speed should be lower - thus higher inference time. This function basically overrides the default behaviour of precomputing the gradients on the forward pass, leading to an increase in speed when used. It's useful for inference because when you're not training your network, you don't need to backpropagate gradients.

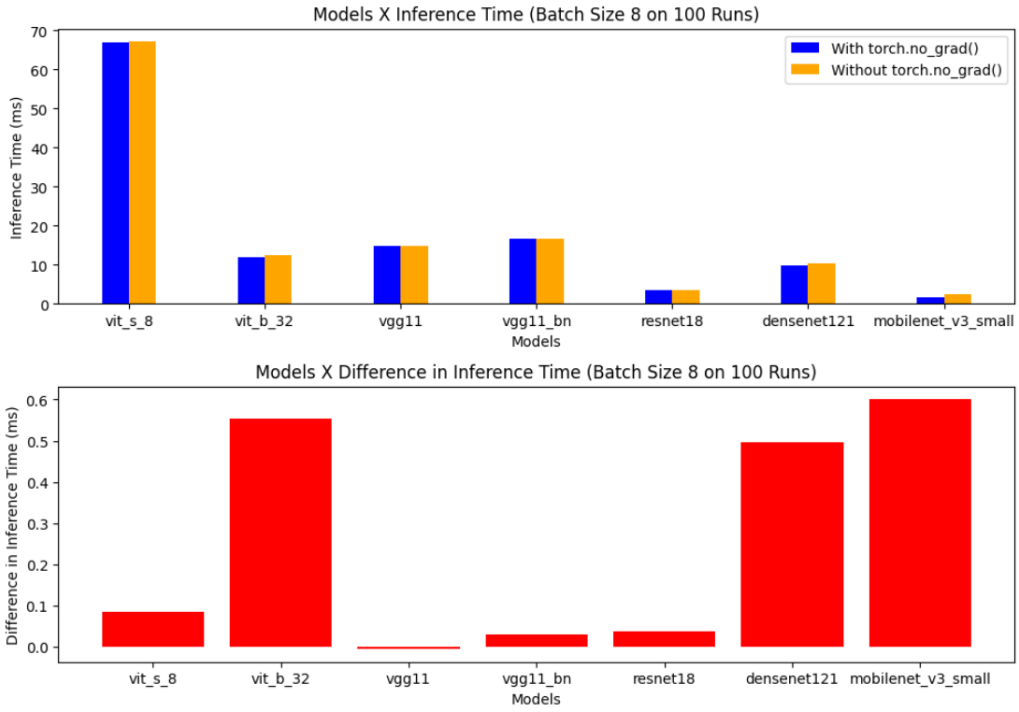

Figure 3: Graph of differences on inference time for models with and without gradient computations for batch size of 8 on 100 runs.

1.1.c

With `torch.no_grad()` the `vRAM` usage should be lower. Since we are not computing and storing the gradients for backpropagation, we save thousands of `Mega Bytes` of memory. Note that this experiment was ran with a batch size of 16 to because of my `GPU` limitations.

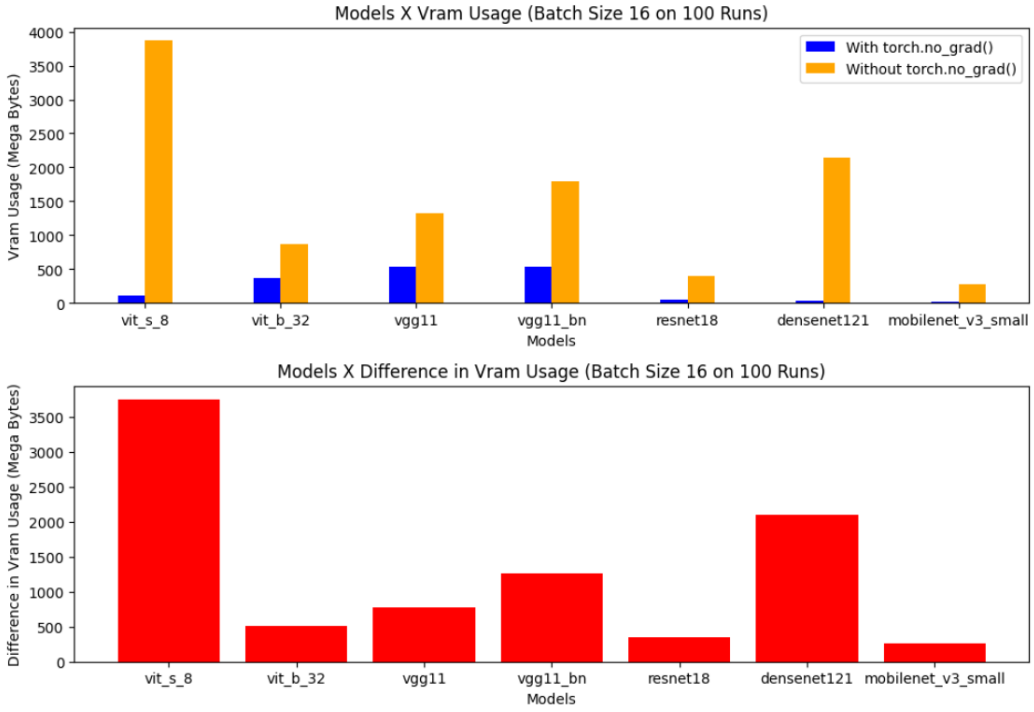

Figure 4: Graph of differences on vRAM usage for models with and without gradient computations for batch size of 16 on 100 runs.

Question 1.2

1.2.a

Adapting the `ResNet-18` model as requested and retraining only the last layer on CIFAR-100 dataset with the default parameters, I got test accuracy of 58.39%.

1.2.b

Augmentations help because they assist the model on generalizing better, making it learn how to understand images rather than memorizing the dataset. Even if we are only training the last layer, our pretrained model has a robust feature space representation of images and thus we can still learn to better generalize.

Augmentation	**Test Accuracy (%)**
Gaussian Noise	3.90
Random Rotation	42.20
Random Perspective	58.63
Color Jitter	4.01
Horizontal Flip	**59.15**
Vertical Flip	55.18
Blur	56.30

Table 2: Table with augmentation transform vs test accuracy.

1.2.c

The first convolutional layer extract low level features like edges. The last convolutional layer extract high level features by combining all the lower level information form the previous layers, identifying patterns for example.

I would rather train the last convolutional layer than the first. The low level features for images are not as different as the high level features for different datasets. So the edge information extracted would be similar for example, but the patterns could be completely different.

2.1

2.1.a

As shown in table 3, the model performs much better on CIFAR10 when compared to CIFAR100. This is expected since CIFAR10 has much less classes and hence presents a much less difficult classification problem. It is also interest to notice the robustness of the model, meaning that the performance on the training set is very similar to the performance on the test set for both datasets.

Dataset/Split	Top-1 Accuracy (%)
CIFAR10/Train	88.68
CIFAR10/Test	**88.77**
CIFAR100/Train	63.55
CIFAR100/Test	63.08

Table 3: Table with dataset and its data split vs accuracy for CLIP-8B/32 model.

2.1.b

For the 'red, green or blue' task, I used the prompt 'The image is mostly '. CLIP performed pretty well in terms of describing the most present color in each image, getting all predictions right apart from the bunny - for which I'm not even sure what the correct answer is. However, in terms of probabilities, the model struggled a bit, getting the right color for most of the images only by a very subtle margin.

The same happens to the 'human-made or nature' task, for which I used the prompt 'This is '. CLIP performed really well, getting only the bunny image wrong and even making me think philosophically about whether the human face picture would be 'human-made' or 'nature'. In terms of probabilities, we can notice once again that the model struggles, getting the predictions right or wrong by a very small margin.

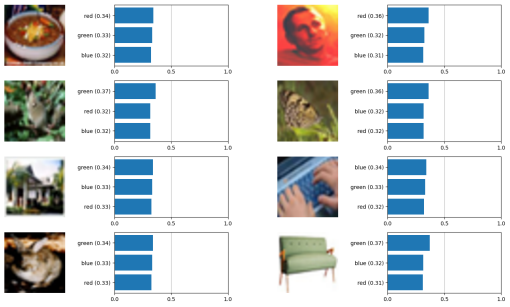

Figure 5: Model's 'red, green or blue' prediction for eight random images from CIFAR100.

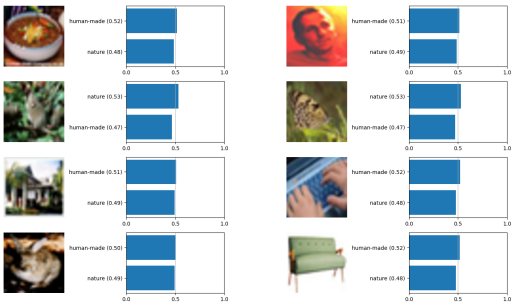

Figure 6: Model's 'human-made or nature' prediction for eight random images from CIFAR100.

2.2

From table 4, it is clear that both the visual prompt techniques can help increasing a bit the top-1 classification accuracy of the model for CIFAR10 dataset. For CIFAR100 dataset, in turn, only fixed patch prompt helped. The phenonmenon of the fixed patch prompt helping a bit more than the padding patch prompt is not expected since padding has a much larger amount of parameters. It is probably related to the number of epochs required to train the padding prompt. Since it adds much more noise to the image, it would need more epochs to make it increase accuracy of the model even further.

Finally, it is important to notice that the default prompt size for fixed patch prompting was 1, which means we have reduced even further the capabilities of the fixed patch prompting. This is worth noting because the experiment is probably not a completely fair comparison of the prompts.

Dataset/VPT/Split	**Top-1 Accuracy (%)**
CIFAR10/Fixed/Validation	88.62
CIFAR10/Fixed/Test	**89.30**
CIFAR10/Padding/Validation	88.62
CIFAR10/Padding/Test	88.96
CIFAR100/Fixed/Validation	**64.65**
CIFAR100/Fixed/Test	64.34
CIFAR100/Padding/Validation	61.32
CIFAR100/Padding/Test	61.19

Table 4: Table with dataset, a type of visual prompt and the data split vs accuracy for CLIP-8B/32 model.

Figure 7: Visual fixed patch ($p = 30$) prompt for sanity check.

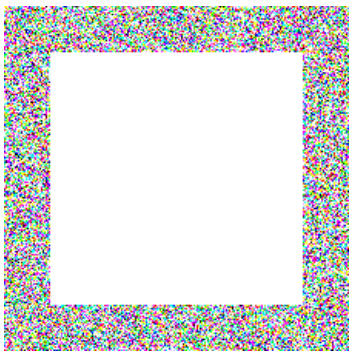

Figure 8: Visual padding patch ($p = 30$) prompt for sanity check.

2.3

The increase in accuracy is significantly higher with deep prompts when compared to visual prompts. While visual prompts, in the best case scenario, increased accuracy in 0.5% to 1%, deep prompts increased accuracies in 3% to 6% in all datasets and splits, regardless of the injection layer.

Injecting prompts on early transformer blocks leads to less change in the model's performance. This is because these layers are responsible for processing low level features, so more generic image and relational features, thus injecting learnable prompts here will probably have less effect on the final outcome.

Prompts on later layers of the network, in turn, can learn to activate more high level features, leading to more specific understanding of the domain. Having said that, prompts on late layers of the network can also lead to overfitting, since they already represent very high level information specific to the training dataset.

These phenomena are pretty well described by table 5. The deeper the prompt injection, the higher the increase in performance, but only up to a certain layer. After this layer, validation and test accuracies start decreasing and the difference between them increase, indicating the model is probably overfitting.

Dataset/Injection Layer/Split	Top-1 Accuracy (%)
CIFAR10/Layer0/Validation	93.19
CIFAR10/Layer0/Test	93.19
CIFAR10/Layer3/Validation	**93.29**
CIFAR10/Layer3/Test	93.26
CIFAR10/Layer8/Validation	92.17
CIFAR10/Layer8/Test	91.76
CIFAR100/Layer0/Validation	69.72
CIFAR100/Layer0/Test	68.91
CIFAR100/Layer3/Validation	**69.94**
CIFAR100/Layer3/Test	68.29
CIFAR100/Layer8/Validation	66.40
CIFAR100/Layer8/Test	65.65

Table 5: Table with dataset, the layer of deep prompt injection and the data split vs accuracy for CLIP-8B/32 model.

2.4

From table 6, it is clear that the CLIP model handles the noisy test data much better than the fine-tuned ResNet does. This is a strong indication that CLIP model is more robust, hence learns more describing and representative features and generalizes better. Additionally considering that fine-tuning requires optimization on the model-space, not on the data-space, it is quite possible that the fine-tuned ResNet oeverfitted to the small and noise-free set of images we used.

Among the CLIP models, the visual prompt with a fixed patch seems to perform better.

The below results were computed with noisy test data and:

1. Fixed patch prompt size: 1
2. Padding prompt size: 30
3. Deep prompt: 4 prompts on layer 3.

Dataset/Model	Top-1 Accuracy (%)
CIFAR10/ResNet	46.91
CIFAR10/CLIP + Fixed Patch	**88.08**
CIFAR10/CLIP + Padding	79.54
CIFAR10/CLIP + Deep Prompt	84.46
CIFAR100/ResNet	24.84
CIFAR100/CLIP + Fixed Patch	**61.11**
CIFAR100/CLIP + Padding	45.51
CIFAR100/CLIP + Deep Prompt	54.38

Table 6: Table with dataset and the model vs accuracy on test data with noise.

2.5

From table 7 it is pretty clear that, regardless of the data we fine-tuned our models on, the accuracies drop for test dat on both CIFAR10 and CIFAR100. This is expected since we are adding more classes that the models were not fine-tuned on.

It is also quite natural that CIFAR100 fine-tuned models will suffer less when predicting on the CIFAR100 test data, since the number of additional classes they haven't seen during training represents only 10% of the total amount of classes. On the other hand, for the CIFAR10 models, the amount of possible classes grew more than 10 times what they were trained on, which influences their accuracy on the CIFAR10 dataset much more.

This is shown in table 7 where one can see that the accuracy drop from what we had in previous exercises for the column 'CIFAR100 (%)' for CIFAR100 models is quite modest when compared to the drops of column 'CIFAR10 (%)' for the CIFAR10 models.

Finally, the table also makes it clear that the models trained on CIFAR100 tend to do better predictions on CIFAR10 than the models trained on CIFAR10 do on CIFAR100. There's an exception for the model trained on CIFAR10 with Fixed Patch, which got solid results for the test data on CIFAR100. Apart from this commented exception, it is quite natural and explicit on table 7 that models trained on one dataset perform on the other dataset much worse than what they could.

Fine-tune Dataset + Pompt	**CIFAR10 (%)**	**CIFAR100 (%)**	**Mean Top-1 Accuracy (%)**
CIFAR10 + Fixed Patch	74.55	**62.07**	**68.31**
CIFAR10 + Padding	78.7	46.4	62.55
CIFAR10 + Deep Prompt	**90.91**	29.56	60.24
CIFAR100 + Fixed Patch	**75.20**	62.64	**68.92**
CIFAR100 + Padding	63.37	61.96	62.67
CIFAR100 + Deep Prompt	68.18	**68.25**	68.22

Table 7: Table with dataset in which the model was fine-tuned evaluated on test data from both datasets.

2.6

Robustness Against Distributional Shifts

In terms of adapting to changes in the data distribution, there can be some differences between the approaches. Full fine-tuning, if given enough training data and time, allows for a more comprehensive training of the model, which probably leads to better results. It is not always the case that we have both training data and time resources though.

The linear probing, in turn, only trains one last layer before doing predictions, hence it very much depends on the robustness of the pretrained model to lead to good results. Therefore this method is much less likely to well adapt to distributional shifts on the data.

Finally, for the prompting approach, since we are directly changing the data, not the parameters of the model, we have more control over the inputs of the network. This means we have more space to adapt the data towards a distribution for which our pretrained model is well prepared. Even though we maintain the simplicity and the fewer parameters we already had for linear probing when compared to full fine-tuning, this approach has the disadvantage of requiring some prompting design which can always be a difficult task.

Computation Resources Required

There are huge differences between the approaches in terms of computational resources. The full fine-tunning model would require us to retrain a full neural network, just with a well selected starting point. Depending on the network architecture and thus the number of parameters, this would require both a lot of time to train and a lot of data to make sure we are not overfitting.

The linear probing, in turn, requires much less parameters since we are only training one full conected layer on top of a frozen pretrained network. This approach thus requires much less time to train and would probably already yield a better model with much less data than the previous approach.

Finally the prompting approach take the training from the model-space to the data-space which can lead reductions in the number of parameters comparing to the previous approach, but most importantly let us train with much less data. Having said that, it requires us to backpropagate the gradients throughout the whole network until we reach the prompt, a much more computational expensive operation than in the last approach for which we only compute gradients for the last (and only trainable) layer.

Memory Requirements

There can also be huge differences between approaches in terms of memory requirements. The full fine-tuning approach requires much more memory usage since we have to compute the gradients for

all the nodes in the network.

For both prompting and linear probing approaches, it very much depends on the chosen parameters. In both cases, we are only storing the gradients for one specific layer of parameters. So the difference between these two depends much more on the number of parameters used to describe this single trainable layer, either the prompt or a fully connected layer.

It is important to notice that, regardless of the transfer learning approach, the model architecture can also influences a lot memory usage.

Overall Performance on the Downstream Task

The performance is probably the hardest metric for which to compare the approaches. That is because it very much depends on the resources you have.

A full fine-tuning can lead to a much better model with sufficient time and sufficiently large dataset, even though the 'sufficient' amount of time and data is very hard to tell. Since normally, both time and data are either scarce, expensive or both, this is probably not the transfer learning approach to use. With not enough time or data, this approach can yield terrible models.

Linear probing is probably the more classic way of doing transfer learning and with not that much training and neither that much data, it can already yield a very good model for the a task. Of course it very much depends on the pretrained model you are using, but this is true for any transfer learning approach.

Finally, prompting is the most modern way of doing transfer learning and has been giving really good results. This method requires even less data to yield models with good performance. Not only that, but it also does optimization in an easier to visualize way, which helps understanding how the network is learning and adjusting to the new task.

3 Graph Neural Networks

3.1

3.1.a

First graph:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Second graph:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Third graph:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

3.1.b

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} 2 & 0 & 1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 0 & 1 & 0 \\ 0 & 1 & 0 & 2 & 0 & 1 \\ 1 & 0 & 1 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 & 0 & 2 \end{bmatrix} \end{array}$$

3.1.c

A^n represent the number of ways to go from one node to another one in n steps. So in our specific case, $(A^2)_{i,j}$ represents how many ways I have to go from node i to node j in 2 steps, and in a generic case, $(A^n)_{i,j}$ represents how many ways I have to go from node i to node j in n steps.

3.2

3.2.a

The structural information is incorporated by the summation $\sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|}$ since we use the average of embedding of neighbors of a node v as a component to compute its embedding in the next layer $h_v^{(l+1)}$.

3.2.b

Equation 2 and equations 3 are very similar, and we can understand equation 2 almost as a special case of what is described by equations 3. Comparing both, we have that $message(h_v^l, h_u^l, e_{uv})$ in equations 3 can be understood as $W^l \frac{h_u^l}{|N(v)|}$ in equation 2, with the difference being the nonlinearity of the message function whereas what is described in equation 2 is a linear transformation of the embeddings. In turn, $update(h_v^l, m_v^{l+1})$ in equation 3 can be understood as $\sigma\left(m_v^{l+1} + B^l h_v^l\right)$ in equation 2.

In that sense, the formulation of a GCN can be understood as a message-passing neural network: message formulation based on the node embeddings we have ($W^l \frac{h_u^l}{|N(v)|}$), message aggregation considering only the neighbors of the node v we are interested in ($\sum_{u \in N(v)} W^l \frac{h_u^{(l)}}{|N(v)|}$), and finally update of embeddings of the node v ($\sigma\left(W^l m_v^{l+1} + B^l h_v^l\right)$).

For the difference between the methods, it is clear that equations 3 offer more flexibility to the model, letting us define different message formulations and update functions, including adding nonlinearities to the message function.

3.3

3.3.a

To go from index notation to matrix notation, it is easier to introduce matrices step by step. So we start by rewriting the message aggregation part of the equation ($\sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|}$) in matrix form.

We can use the row v of the adjacency matrix, denoted by $A_v \in \mathbb{R}^{1 \times |V|}$, to select from $H^{(l)} \in \mathbb{R}^{|V| \times d^{(l)}}$ only the embeddings of nodes u that are in the neighborhood of v, so $A_v H^{(l)} = \sum_{u \in N(v)} h_u^{(l)}$. To compute the sum for each node, we simply do $AH^{(l)}$.

We can now use the degree matrix D to get the proper normalization. Since it is a diagonal matrix, it's inverse is given by the inverse of each diagonal entry, so:

$$D_{vu}^{-1} = \begin{cases} \frac{1}{|N(v)|}, & \text{if } v = u \\ 0, & \text{otherwise} \end{cases}$$

Hence, by multiplying the inverse of the degree matrix, $D^{-1} \in \mathbb{R}^{|V|\times|V|}$ with the matrices we had before, $AH^{(l)} \in \mathbb{R}^{|V|\times d^{(l)}}$, we normalize each row by $|N(v)|$, so $D^{-1}AH^{(l)} \in \mathbb{R}^{|V|\times d^{(l)}}$.

Now we just need to multiply the parameter matrices $W^{(l)}, B^{(l)} \in \mathbb{R}^{d^{(l+1)}\times d^{(l)}}$ we had before to get to:

$$H^{(l+1)} = \sigma\left(D^{-1}AH^{(l)}(W^{(l)})^T + H^{(l)}(B^{(l)})^T\right) \in \mathbb{R}^{|V|\times d^{(l+1)}}$$

A matrix containing in each row v the embedding of layer $l+1$ of the node v.

3.3.b

We use the average function to aggregate over neighbor nodes.

3.3.c

No, it is not always possible to rewrite the update formula in the matrix form. It very much depends on the complexity of the aggregation function. One interesting example is the median function, a nonlinear operation that simply cannot be represented by a matrix multiplication. In this case, for each node we would have to sort every feature of neighbors' embeddings to compute the aggregation function value.

3.4

There are a few problems with this approach:

1. The first and major problem would be structural. A standard feed-forward neural network always expect data of a fixed size - the same amount of features per datapoint. However, depending on the number of nodes in a graph, we can have adjacency matrices of different sizes. Because of that, we would only be able to feed the network with graphs of the order, creating a problem that we would come across multiple times with graphs as data. This is the case where a standard NN would fail completely.

2. Another very much problematic aspect is the multiplicity of representations of a graph. The same graph can have multiple adjacency matrices depending on how you name the nodes. This means that the same graph can have multiple adjacency + node embeddings representations, which would confuse a standard neural network a lot and make the training process much harder.

3. The last important fact, connected to the previous one, is that standard neural networks don't explore some vital graph properties. This particular issue arises from the fact that in a standard NN, in the classification scenario, we try to find distributions for features that better describe each one of the classes. In a graph, however, the relationship between nodes is as important as the features' distributions. So not inherently incorporating how nodes are connected to the network's structure adds much more complexity to the training process again.

3.5

3.5.a

The transformer model has a clear parallel to GNNs. We can understand the tokens as the nodes of a graph and the attention mechanism as the definition of the edges. In that sense, a transformer defines a complete graph and the attention mechanism is responsible for iteratively weighting the edges to express stronger and weaker relationship between tokens. It is also interesting to notice that even the

node representation in the network is similar, since we don't use the words themselves in a transformer, but embeddings of them as the tokens.

We can even extend this analogy to message-passing networks. The weighted summation of tokens provided by the attention mechanism works as the aggregation function and then the activations provide the nonlinearity we are going to use to properly update the embeddings of the tokens.

3.5.b

The transformer model does that by adding the locality information on the tokens. The process is done by adding the positional encoding to the tokens' embedding on both the encoder and decoder part of the module. In this way, even though we make use of a permutation invariant architecture, we incorporate ordering beforehand.