Assignment 2. Convolutional, Recurrent, and Graph Neural Networks

University of Amsterdam – Deep Learning Course 1

November 19, 2021

The deadline for this assignment is December 3, 2021.

In this assignment, you will study and implement convolutional, recurrent, and graph neural networks (CNNs, RNNs, GNNs). CNNs are the standard solution for images. RNNs are best suited for sequential data processing, such as a sequence of characters, words, or video frames. Their applications are mostly in neural machine translation, speech analysis, and video understanding. GNNs are specifically applied to graph-structured data, like knowledge graphs, molecules, or citation networks.

The assignment consists of three parts. First, you will get familiar with CNNs and how they transfer to different corruption functions on the test set. In the second part, we look at the theory behind vanilla RNNs and another type of RNN called a Long Short-Term Memory (LSTM) network. Then you will implement an LSTM yourself and test it on generating text. In the final part, you will analyze the forward pass of a graph convolutional neural network and implement your own small GNN. In addition to the coding assignments, the text contains multiple pen and paper questions which you need to answer. We expect each student to hand in their code individually and submit their answers via Ans.

1 Convolutional Neural Networks **(Total: 25 points)**

1.1 Vanilla CNN

Although standard Multilayer Perceptrons (MLP) achieve good results in some tasks as seen in the previous assignment, they are not as scalable as eventually needed in practice. The disadvantage is that the number of total parameters can grow very quickly, especially when having high dimensional inputs like images. This is inefficient because there is redundancy in such high dimensions. To address the issue, Convolutional Neural Networks (CNNs) have been proposed. Using a shared set of weights at each layer, CNNs are more efficient than MLPs. The intrinsic feature of images, which is translation invariance, can be employed to boost the performance of downstream tasks.

Question 1.1 (10 points)

You are asked to refer to the Jupyter notebook that has been attached to this assignment. In each section of the notebook, you are asked to implement the necessary parts of common layers in a CNN from scratch using Numpy functions. This helps you understand the underlying forward and backward operations that exist in the PyTorch convolution functions.

Hint: you are not required to implement it in an efficient matrix-optimized form but can use for-loops to simplify the implementation.

1.2 Modern Architectures

After the proposal of CNNs, different CNN-based architectures have been introduced, which usually differ in depth, number of filters, ways of applying filters to the input information flow, and etc. Some of the most famous architectures are VGG-11 [1], ResNet-32 [2], and DenseNet-121 [3]. The latter two we have already implemented and trained in Tutorial 5. Although state-of-the-art (SOTA) CNN-based methods have achieved even better human performance during the last few years, the amount of transferability, generalization, and robustness of the learned features are still a matter of debate. In this part, by conducting simple experiments, you will scrutinize the learned features more to understand their limitations better. Please refer to the attached python file in which four common corruptions with different severities have been defined. For instance, we experiment with different levels of Gaussian noise and check how much this influences the CNNs prediction accuracy. You have to report the corruption error of each of the above-mentioned architectures using the following criteria:

$$\mathrm{CE}_c^f = \frac{(\sum_{s=1}^{5} E_{s,c}^f)}{(\sum_{s=1}^{5} E_{s,c}^{\text{ResNet-18}})}, \tag{1}$$

$$\mathrm{RCE}_c^f = \frac{(\sum_{s=1}^{5} E_{s,c}^f - E_{\text{clean}}^f)}{(\sum_{s=1}^{5} E_{s,c}^{\text{ResNet-18}} - E_{\text{clean}}^{\text{ResNet-18}})}, \tag{2}$$

where E_{clean}^f denotes the clean dataset top-1 error rate of architecture f, and $E_{s,c}^f$ shows the error after applying severity s and corruption c to the test dataset. Since different corruption functions impose different severities, which are all aggregated into a summation to show the general robustness of the models, normalizing them with respect to a standard factor is necessary. Therefore, here, all the calculations are normalized based on the performance of **ResNet-18** when various corruptions are applied. All the experiments will be performed on CIFAR-10 to keep the computational cost on a reasonable scale.

Question 1.2 (15 points)

(a) (8 points) Implement the training and testing of a ResNet-18 architecture in the provided python templates. Report the test accuracies of the ResNet-18 on the four provided corruption functions `Gaussian noise`, `Gaussian blur`, `Contrast reduction`, and `JPEG compression` over different severities (1 to 5), including the test accuracy on the original test dataset. Plot the results with accuracy on the y-axis, severity on the x-axis, and different lines for the different corruption functions. Shortly, describe the trends you observe.

(b) (7 points) Redo the experiment for the following network architectures:

- VGG-11
- VGG-11 with Batch Normalization
- ResNet-34
- DenseNet-121

Report the results in terms of the previously defined metrics, CE and RCE, for the four different corruption functions. Explain the trend you observe when comparing network architectures, deeper models, and using Batch Normalization.

Hint: This question requires a considerable amount of computation, and thus it is strongly recommended deploying the code on Lisa for this. The estimated runtime of all models together is about 6hrs, so ***do not wait until the last minute*** *to run these models. Use the provided 'debug' model for faster debugging on CPU.*

2 Recurrent Neural Networks **(Total: 35 points)**

2.1 Vanilla RNNs

The vanilla RNN is formalized as follows. Given a sequence of input vectors $\mathbf{x}^{(t)}$ for $t = 1, \dots T$, the network computes a sequence of hidden states $\mathbf{h}^{(t)}$ and a sequence of output vectors $\mathbf{p}^{(t)}$ using the following equations for time steps $t = 1, \dots, T$:

$$\mathbf{h}^{(t)} = \tanh(\mathbf{W}_{hx}\mathbf{x}^{(t)} + \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{b}_h) \tag{3}$$

$$\mathbf{p}^{(t)} = \mathbf{W}_{ph}\mathbf{h}^{(t)} + \mathbf{b}_p \tag{4}$$

As you can see, there are several trainable weight matrices and bias vectors. $\mathbf{W}_{hx}$ denotes the input-to-hidden weight matrix, $\mathbf{W}_{hh}$ is the hidden-to-hidden (or recurrent) weight matrix, $\mathbf{W}_{ph}$ represents the hidden-to-output weight matrix and the $\mathbf{b}_h$ and $\mathbf{b}_p$ vectors denote the biases. For the first time step $t = 1$, the expression $\mathbf{h}^{(t-1)} = \mathbf{h}^{(0)}$ is replaced with a special vector $\mathbf{h}_{init}$ that is commonly initialized to a vector of zeros. The output value $\mathbf{p}^{(t)}$ depends on the state of the hidden layer $\mathbf{h}^{(t)}$ which in its turn depends on all previous state of the hidden layer. Therefore, a recurrent neural network can be seen as a (deep) feed-forward network with shared weights.

To optimize the trainable weights, the gradients of the RNN are computed via back-propagation through time (BPTT). The goal is to calculate the gradients of the loss $\mathcal{L}$ with respect to the model parameters $\mathbf{W}_{hx}, \mathbf{W}_{hh}$ and $\mathbf{W}_{ph}$ (biases omitted). Similar to training a feed-forward network, the weights and biases are updated using SGD or one of its variants. Different from feed-forward networks, recurrent networks can give output predictions $\hat{\mathbf{y}}^{(t)}$ at every time step. In this assignment the outputs will be given by the softmax function, *i.e.* $\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{p}^{(t)})$. For training, we compute the standard cross-entropy loss at each time step t:

$$\mathcal{L}^{(t)} = -\sum_{k=1}^{K} \mathbf{y}_k^{(t)} \log \hat{\mathbf{y}}_k^{(t)} \tag{5}$$

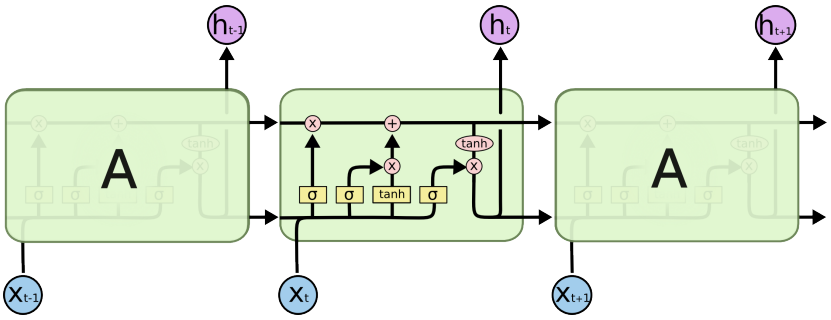


Figure 1. A graphical representation of LSTM memory cells (Olah, 2015 [4])

Where k runs over the number of classes. In this expression, $\mathbf{y}$ denotes a one-hot vector of length K containing true labels. The gradient w.r.t. $\mathbf{W}_{ph}$ at time step t can be expressed as follows:

$$\frac{\partial \mathcal{L}^{(t)}}{\partial \mathbf{W}_{ph}} = \frac{\partial \mathcal{L}^{(t)}}{\partial \hat{\mathbf{y}}^{(t)}} \frac{\partial \hat{\mathbf{y}}^{(t)}}{\partial \mathbf{p}^{(t)}} \frac{\partial \mathbf{p}^{(t)}}{\partial \mathbf{W}_{ph}} \tag{6}$$

For the gradients w.r.t. $\mathbf{W}_{hh}$ and $\mathbf{W}_{hx}$, we need to sum up the contributions of each time step to the gradient. In case of the weight matrix $\mathbf{W}_{hh}$, this leads to the following expression:

$$\frac{\partial \mathcal{L}^{(t)}}{\partial \mathbf{W}_{hh}} = \frac{\partial \mathcal{L}^{(t)}}{\partial \hat{\mathbf{y}}^{(t)}} \frac{\partial \hat{\mathbf{y}}^{(t)}}{\partial \mathbf{p}^{(t)}} \frac{\partial \mathbf{p}^{(t)}}{\partial \mathbf{h}^{(t)}} \sum_{t'=0}^{t} \left(\prod_{j=t'+1}^{t} \frac{\partial \mathbf{h}^{(j)}}{\partial \mathbf{h}^{(j-1)}} \right) \frac{\partial \mathbf{h}^{(t')}}{\partial \mathbf{W}_{hh}} \tag{7}$$

Note that the product $\prod_{j=t'+1}^{t} \partial \mathbf{h}^{(j)} / \partial \mathbf{h}^{(j-1)}$ for $t' = t$ equals 1 by convention. The gradient w.r.t. $\mathbf{W}_{hx}$ is expressed accordingly.

Question 2.1 (4 points)

Recurrent neural networks can be trained using backpropagation through time. Similar to feed-forward networks, the goal is to compute the gradients of the loss w.r.t. $\mathbf{W}_{ph}$, $\mathbf{W}_{hh}$ and $\mathbf{W}_{hx}$.

(a) (2 points) Expand the expression of the gradient $\frac{\partial \mathcal{L}^{(t)}}{\partial \mathbf{W}_{hh}}$ for time step $t = 3$, i.e. write out the sum and product in terms of the variables that appear in Equation 7 and simplify where possible. In your solution, every time-index should be replaced by a number.

(b) (2 points) As can be seen from Equation 7, the gradient w.r.t. $\mathbf{W}_{hh}$ depends on all past time steps, whereas the gradient w.r.t. $\mathbf{W}_{ph}$ only depends on the current time step. Study the gradient w.r.t. $\mathbf{W}_{hh}$ and explain what problems might occur when training this recurrent network for a large number of time steps. You can use your solution from (a) to support your explanations.

2.2 Long Short-Term Memory (LSTM) network

Training a vanilla RNN for remembering its inputs for an increasing number of time steps is difficult. The problem is that the influence of a given input on the hidden layer (and therefore on the output layer), either decays or blows up exponentially as it unrolls the network. In practice, the *vanishing gradient problem* is the main shortcoming of vanilla RNNs. As a result, training vanilla RNNs to consistently learn tasks containing delays of more than ~ 10 time steps between relevant input and target is difficult. To overcome this problem, many different RNN architectures have been suggested. The most widely

used variant is the Long Short-Term Memory networks (LSTMs). An LSTM (Figure 1) introduces several gating mechanisms to improve gradient flow for a more stable training procedure. Before continuing, we recommend reading the following blog post to get familiar with the LSTM architecture: **Understanding LSTM Networks** [4].

In this assignment we will use the following LSTM definition:

$$\mathbf{g}^{(t)} = \tanh(\mathbf{W}_{gx}\mathbf{x}^{(t)} + \mathbf{W}_{gh}\mathbf{h}^{(t-1)} + \mathbf{b}_g) \tag{8}$$

$$\mathbf{i}^{(t)} = \sigma(\mathbf{W}_{ix}\mathbf{x}^{(t)} + \mathbf{W}_{ih}\mathbf{h}^{(t-1)} + \mathbf{b}_i) \tag{9}$$

$$\mathbf{f}^{(t)} = \sigma(\mathbf{W}_{fx}\mathbf{x}^{(t)} + \mathbf{W}_{fh}\mathbf{h}^{(t-1)} + \mathbf{b}_f) \tag{10}$$

$$\mathbf{o}^{(t)} = \sigma(\mathbf{W}_{ox}\mathbf{x}^{(t)} + \mathbf{W}_{oh}\mathbf{h}^{(t-1)} + \mathbf{b}_o) \tag{11}$$

$$\mathbf{c}^{(t)} = \mathbf{g}^{(t)} \odot \mathbf{i}^{(t)} + \mathbf{c}^{(t-1)} \odot \mathbf{f}^{(t)} \tag{12}$$

$$\mathbf{h}^{(t)} = \tanh(\mathbf{c}^{(t)}) \odot \mathbf{o}^{(t)} \tag{13}$$

$$\mathbf{p}^{(t)} = \mathbf{W}_{ph}\mathbf{h}^{(t)} + \mathbf{b}_p \tag{14}$$

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{p}^{(t)}). \tag{15}$$

In these equations $\odot$ is element-wise multiplication and $\sigma(\cdot)$ is the sigmoid function. The first six equations are the LSTM's core part, whereas the last two equations are just the linear output mapping. Note that the LSTM has more weight matrices than the vanilla RNN. As the forward pass of the LSTM is relatively intricate, writing down the correct gradients for the LSTM would involve a lot of derivatives. Fortunately, LSTMs can easily be implemented in PyTorch, and automatic differentiation takes care of the derivatives.

Question 2.2 (6 points)

The LSTM extends the vanilla RNN cell by adding four gating mechanisms. Those gating mechanisms are crucial for successfully training recurrent neural networks.

(a) (4 points) The LSTM has an *input modulation gate* $\mathbf{g}^{(t)}$, *input gate* $\mathbf{i}^{(t)}$, *forget gate* $\mathbf{f}^{(t)}$ and *output gate* $\mathbf{o}^{(t)}$. For each of these gates, write down a brief explanation of their purpose; explicitly discuss the non-linearity they use and motivate why this is a good choice. (up to 3 sentences per gate)

(b) (2 points) Given the LSTM cell as defined by the equations above, let $\mathbf{x} \in \mathbb{R}^{T \times d}$ be an input sample where T denotes the sequence length and d is the feature dimensionality. Write down the formula for the *total number* of trainable parameters in the LSTM cell as defined above, in terms of N_{hidden}, N_{input} and N_{output} for the dimensionality associated with the hidden cell state, the input and the output respectively.

2.3 LSTMs in PyTorch

In this part of the assignment, you will implement your own LSTM module in PyTorch. In the next part of the assignment, you will use this implementation on text data, where the LSTM learns to predict the next character in a sentence. The inputs to the LSTM will therefore be *sequences of characters*. Since characters are categorical and do not have a trivial order, they can not simply be represented as integers. It is common to encode categorical input data with the help of *embeddings* before passing the data through the network. For an explanation of embeddings, see below. For this task, it is not necessary yet to set hyperparameters apart from the initialization of the weights, which is specified below.

Embeddings

As the network does not accept actual characters ('e', 'a', or whitespace) as inputs, we need to transform these characters to numerical values. As explained previously, one-hot encoding is often preferred over simply (and somewhat arbitrarily) mapping each input to a certain number, e.g., $a = 1$, $b = 2$, etc. The reasoning for this was that the to-be-encoded characters are unrelated. (Certainly $b \neq 2 \cdot a$, so this mapping would indeed not make sense.)
However, there are two more things to take into consideration.

1. The characters may not be *completely* unrelated: We may find a "more similar" to e (both are vowels) than to k or x. Using one-hot-encoding does not allow these possible differences in similarity.
2. If we have many different characters that are all mapped using one-hot-encoding, the input dimension is also large, requiring bigger (thus slower, less efficient, or harder-to-train) input modules.

For these reasons, we use more general maps called **embeddings**. From a broader perspective, note that one-hot-encoding is a way of mapping the N possible categories to an N-dimensional space, whereas assigning numbers is a way of mapping the N categories to a 1-dimensional space. We can now also think of mapping the N possible categories to some M-dimensional space, where usually $M \leq N$. Instead of deciding the actual map manually, we allow the network to train the map, discovering the most practical representation on the go. One can think of starting using N-dimensional one-hot-encoded vectors and mapping these to an M-dimensional space using a matrix $W_{\text{embed}} \in \mathbb{R}^{M \times N}$, to end up with an M-dimensional input vector per character for the neural network.

Weight initialization

In many cases, LSTMs are simply initialized with small random weights. For this assignment, you should initialize all weights uniformly between $-1/\sqrt{N_{hidden}}$ and $+1/\sqrt{N_{hidden}}$. Additionally, you should add a value of 1 to the bias of the forget gate. The reason for this is that for learning long-term dependencies, it is good practice to initialize the bias of the forget gate to a larger value, such that the model starts off with remembering old states and learns what to forget (rather than vice versa).

Question 2.3 (7 points)

Implement an LSTM network as specified by Equations 8 - 13 above in the `LSTM`-class of the file `part2/model.py`. You do not need to implement the linear output mapping yet (i.e. the computation of $\mathbf{p}^{(t)}$), since it does not belong to the LSTM's core part. For the text generation in Question 2.4, you will use embeddings to represent each character of the input sequence (for a general explanation, see above). Therefore, you can assume the input to your LSTM module to be the *embedding representation* of text sequences. The sequence length and batch size are variable.

You are required to implement the model without any high-level PyTorch functions. Specifically, you *are* allowed to use `nn.Parameter` and the PyTorch non-linearity functions (e.g. `torch.sigmoid`), but you are *not* allowed to use `nn.Linear`, `nn.LSTM` or `nn.LSTMCell`. For the initialization of the weights, see the specifications above. You do not need to implement the *backward* pass yourself, but instead, you can rely on automatic differentiation.

2.4 Recurrent Nets as Generative Model

In this part, you will use your implementation of the LSTM for the generation of text. The LSTM will learn the local structure in the text by training to predict the next character in a sentence. You will train a one-layer LSTM on sentences from a book and use the model to generate new text. Before starting, we recommend reading the blog post **The Unreasonable Effectiveness of Recurrent Neural Networks**.

Given a training sequence $\mathbf{x} = (x^{(1)}, \ldots, x^{(T)})$, a recurrent neural network can use its output vectors $\mathbf{p} = (p^{(1)}, \ldots, p^{(T)})$ to obtain a sequence of predictions $\hat{\mathbf{y}}^{(t)}$. When working with text input, each element in the prediction sequence is a *distribution* over the next character given the text available so far. As an example, for the training sequence "Hello world!", the first element of the prediction sequence $\hat{\mathbf{y}}^{(1)}$ would be a distribution over all possible start characters, of which only one dimension resembles the probability p('H'). Consequently, the third element $\hat{\mathbf{y}}^{(3)}$ is a distribution over the next character given the past input characters $(x^{(1)}, x^{(2)})$ = 'He', where one dimension of the prediction resembles p('l'|'He'). Since the network predicts future characters based only on the past, it is an *autoregressive* model.

The network is trained using the total cross-entropy loss, which can be computed by averaging over all time steps using the target labels $\mathbf{y}^{(t)}$.

$$\mathcal{L}^{(t)} = -\sum_{k=1}^{K} \mathbf{y}_k^{(t)} \log \hat{\mathbf{y}}_k^{(t)} \tag{16}$$

$$\mathcal{L} = \frac{1}{T} \sum_{t} \mathcal{L}^{(t)} \tag{17}$$

Again, k runs over all possible classes, where K is the vocabulary size (for text, the classes could e.g. be all ASCII characters). In this expression, $\mathbf{y}$ denotes a one-hot vector of length K containing true labels. Using this sequential loss, you can train a recurrent network to make a prediction at every time step.

After training, the LSTM can generate text character-by-character that will look similar to the original text. For this, the network uses its own output as an input for the next prediction. Just like multilayer perceptrons, LSTM cells can be stacked to create deeper layers for increased expressiveness. Each recurrent layer can be unrolled in time. For this task, we will only use one LSTM layer.

For training, you can use a large text corpus, such as publicly available books. We provide several books in the `assets` directory. However, you are also free to download other books. We recommend Project Gutenberg as a good source. Make sure you download the books in plain text (.txt) for easy file reading in Python. We provide the `TextDataset` class for loading the text corpus and drawing batches of example sentences from the text.

The sequence length specifies the length of training sentences, which also limits the number of time steps for backpropagation in time. When setting the sequence length to 30 steps, the gradient signal will never backpropagate more than 30 time steps. As a result, the network cannot learn text dependencies longer than this number of characters.

Teacher Forcing

In the standard RNN setting, the output of a previous module is used as input of the next module. During inference, a "start-of-sequence" token is used as input for the first module, and the network takes it from there, feeding each module with the output of its predecessor. During training, however, we do not have to follow this recipe strictly. In what is known as **Teacher Forcing**, we do not use the (possibly faulty) intermediate outputs of an RNN as inputs, but instead use the ground truth (the correct answer).
We still compute the loss based on those intermediate outputs, but by using Teacher

Forcing, we give the network a fair chance on the subsequent modules to produce the correct output.[1] Note that if we hadn't, the further we go in the sequence of modules, the more likely it is that everything is messed up due to some faulty outputs early on in the recurrence. Such an accumulation of errors makes it difficult for the model to learn.
This is also observed in practice, where models without Teacher Forcing take longer to converge because the first few words need to be correct before successful training on longer sentences can occur.

Question 2.4 a) (8 points)

Study the code and its outputs in `part2/dataset.py` to sample sentences from the book to train with. Also, have a look at the parameters defined in `part2/train.py`. You need to implement the corresponding PyTorch code in `part2/train.py` to make the features work. You may need to tune them depending on your own implementation and chosen dataset. In the file `part2/model.py`, you need to implement the class `TextGenerationModel` and its forward-function.

(a) (8 points) Implement a one-layer LSTM network with 1024 hidden dimensions to predict the next character in a sentence by training on sentences from a book. Use your own implementation from Question 2.3 for the LSTM layer. If you were not able to implement the LSTM yourself, you are allowed to use the PyTorch module `nn.LSTM`. However, if your own implementation of the LSTM is working, using `nn.LSTM` will lead to point deductions. Note that you will need to transform the input into embeddings (use `nn.Embedding`) and pass the output of the LSTM module through a linear output layer if the output layer is not included in your LSTM module.

Train the model on sentences of length $T = 30$ from your book of choice. Define the total loss as the average of cross-entropy losses over all 30 time steps (Equation 17). Plot the loss and accuracy during training, and report all the relevant hyperparameters that you used and shortly explain why you used them (even if they are the default ones). You can use TensorBoard for creating the plots. For convenience, use only a train set; no validation nor test sets required.
Hint: To train the model efficiently, use teacher forcing, as explained above.

[1] One can compare this with an exam question that consists of multiple parts, where the answer of part (a) is needed for part (b), and so on. If you get (a) wrong, you are likely to get the rest of the question wrong as well. However, if you are lucky, the question might say, "If you couldn't find the answer for (a), use: ..." This allows you to complete the rest of the question, even though you do lose points on part (a). With Teacher Forcing, you make sure the network trains successfully on the entire question by always filling in the dots with the correct answer (regardless of whether the network found the correct answer or not).

Question 2.4 b) (5 points)

(b) (5 points) Make the network generate new sentences of length $T = 30$ at three different points during training to show the progress of the model (e.g., after 1 epoch, 5 epochs and at the end of the training). You can do this by randomly setting the first character of the sentence and always picking the token with the highest probability predicted by your model. Store your hidden state in between token generations to prevent doing duplicate calculations. Report 5 text samples with different start characters generated by the network over the various stages of training. Carefully study the text generated by your network. What changes do you observe when the training process evolves? For your dataset, some patterns might be better visible when generating sentences longer than 30 characters. Discuss the difference in the quality of sentences generated (e.g., coherency) for a sequence length of less and more than 30 characters.

Question 2.4 c) (5 points)

(c) (5 points) In your current implementation, your next character is always chosen by selecting the one with the highest probability, regardless of the rest of the distribution.[a] On the opposite, we could also perform *random sampling*, where we generate an actual sample from the autoregressive distribution learned by the model: this will result in a high diversity of sequences, but they will be meaningless. However, we can interpolate between these two extreme cases by using the outputted distribution together with a *temperature* parameter τ in the softmax:

$$\text{softmax}(\tilde{x}) = \frac{\exp(\tilde{x}/\tau)}{\sum_i \exp(\tilde{x}_i/\tau)}$$

(for details, see Goodfellow *et al.*; Section 17.5.1).

- Explain the effect of the temperature parameter τ on the sampling process.
- Extend your current model by adding the temperature parameter τ to balance the sampling strategy between fully-greedy and fully-random.
- Report generated sentences for temperature values $\tau \in \{0.5, 1.0, 2.0\}$ of your fully trained model. What do you observe for different temperatures? What are the differences with respect to the sentences obtained by greedy sampling?

Note that using one single dataset is sufficient to get full points. However, we encourage you to experiment using different datasets to gain more insight. We suggest starting with relatively small (and possibly with simple language) datasets, such as Grim's fairy tales, so that you can train the model on your laptop easily. If the network needs training for some hours until convergence, it is advised to run training on the SurfSara cluster. Also, you might want to save the model checkpoints now and then, so you can resume training later or generate new text using the trained model.

[a] Such an implementation is also called *greedy sampling*. More generally, a greedy algorithm will make whatever choice seems best at the moment, without taking into account the resulting future state of the algorithm caused by that choice.

3 Graph Neural Networks **(Total: 40 points)**

Make sure to check the UvA Deep Learning Tutorial 7 about GNNs (link) before continuing with this part.

3.1 GCN Forward Layer

Graph convolutional neural networks are widely known architectures used to work with graph-structured data, and a particular version (GCN, or Graph Convolutional Network) was first introduced in https://arxiv.org/pdf/1609.02907.pdf. Consider Eq. 18, describing the propagation rule for a layer in the Graph Convolutional Network architecture to answer the following questions.

$$H^{(l+1)} = \sigma(\hat{A}H^{(l)}W^{(l)}) \tag{18}$$

Where $\hat{A}$ is obtained by:

$$\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}} \tag{19}$$

$$\tilde{A} = A + I_N \tag{20}$$

$$\tilde{D}_{ii} = \sum_j \tilde{A}_{ij} \tag{21}$$

In the equations above, $H^{(l)}$ is the $N \times d$ matrix of activations in the l-th layer, A is the $N \times N$ adjacency matrix of the undirected graph, I_N is an identity matrix of size N, and $\tilde{D}$ is a diagonal matrix used for normalization (you don't need to care about this normalization step; instead, you should focus on discussing Eq. 18). $\tilde{A}$ is the adjacency matrix considering self-connections, N is the number of nodes in the graph, d is the dimension of the feature vector for each node. The adjacency matrix A is usually obtained from data (either by direct vertex attribute or by indirect training). W is a learnable $d^{(l)} \times d^{(l+1)}$ matrix utilized to change the dimension of feature per vertex during the propagation over the graph.

Question 3.1 (3 points)

Describe how this layer exploits the structural information in the graph data, and how a GCN layer can be seen as performing message passing over the graph.

3.2 Relation of Convolutions and Graphs

As a special case of a graph consider a one-dimensional grid with N nodes with periodic boundary (a *circulant graph*): we label the nodes so that the i-th node is connected to the $(i-1 \bmod n)$-th and $(i+1 \bmod n)$-node (Figure 2).

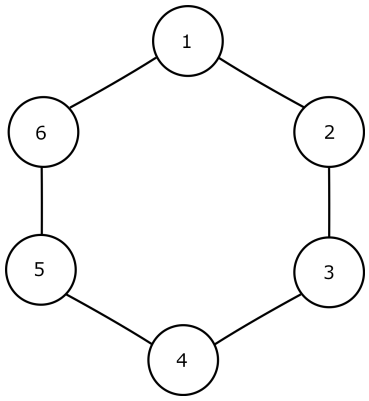


Figure 2. 1D grid with $n = 6$ nodes.

Question 3.2 (2 points)

Write the adjacency matrix of the circular graph with n nodes. (use dots ... but make sure their meaning is not ambiguous)

On this graph, we design the following GNN layer:

$$H^{(l+1)} = F_\theta(H^{(l)}), \tag{22}$$

where

$$F_\theta(H) = \mathbf{C}(\theta)H := \begin{bmatrix} w_0 & w_1 & 0 & \dots & 0 & w_{-1} \\ w_{-1} & w_0 & w_1 & 0 & \dots & 0 \\ 0 & w_{-1} & w_0 & w_1 & 0 & \dots \\ \vdots & & \ddots & \ddots & \ddots & \\ 0 & \dots & 0 & w_{-1} & w_0 & w_1 \\ w_1 & 0 & \dots & 0 & w_{-1} & w_0 \end{bmatrix} \begin{bmatrix} - - - & h_0 & - - - \\ - - - & h_1 & - - - \\ - - - & h_2 & - - - \\ & \vdots & \\ - - - & h_{n-2} & - - - \\ - - - & h_{n-1} & - - - \end{bmatrix}, \tag{23}$$

where $\theta = (w_0, w_1, 0, \dots, 0, w_{-1})$, and h_i is the feature vector of the node i so that H is a $D \times n$ matrix. The general form of a permutation equivariant GNN layer is

$$F(H) = \begin{bmatrix} \phi(h_0, \mathcal{N}_0) \\ \phi(h_1, \mathcal{N}_1) \\ \vdots \\ \phi(h_{n-2}, \mathcal{N}_{n-2}) \\ \phi(h_{n-1}, \mathcal{N}_{n-1}) \end{bmatrix},$$

Where $\mathcal{N}_i$ is the neighbourhood of the node i.

Question 3.3 (6 points)

(a) (2 points) What is $\phi(h_i, \mathcal{N}_i)$ for the GNN layer of equation (23)? Write the explicit form of ϕ as a function of the nodes features h_i.

(b) (2 points) What operation is performed by this GNN layer on a circulant graph with one-dimensional features (H is a column vector)?

(b) (2 points) Using the definition of $\mathbf{C}(\theta)$ provided by equation (23), what would be the action of $F_{\tilde{\theta}}$ for $\tilde{\theta} = (0, 1, 0, \dots, 0)$?

In what follows, you will derive a theorem that is fundamental for many spectral-based convolutional graph neural networks such as GCNs: the (discrete version of the) Convolution Theorem. The theorem states that the Discrete Fourier Transform (DFT) of a convolution of two signals $\mathbf{x} = (x_0, x_1, \dots, x_{n-1})$ is the element-wise product of their DFTs:

$$\mathbf{C}(\theta)\mathbf{x} = \Phi\hat{\mathbf{x}} \odot \hat{\theta}, \tag{24}$$

where $\Phi^*\mathbf{x} = \hat{\mathbf{x}}$, $\Phi\hat{\mathbf{x}} = \mathbf{x}$ are the DFT and inverse DFT respectively (Φ^* is the transpose conjugate of Φ), and $\mathbf{C}(\theta)$ is our circulant matrix with filter θ.

The Discrete Fourier Transform

The DFT and the inverse DFT of a signal $\mathbf{x} = (x_0, x_1, \dots, x_{n-1})$ are defined as

$$\hat{x}_k = \frac{1}{\sqrt{n}} \sum_{u=0}^{n-1} x_u e^{-\frac{2\pi i k u}{n}} \qquad x_u = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} \hat{x}_k e^{\frac{2\pi i k u}{n}}. \tag{25}$$

Despite the seemingly complicated form of the DFT, the idea behind it is rather simple: the complex numbers $e^{-\frac{2\pi i k u}{n}}$ represent phases and frequencies, and the magnitude of k-th entry of the transformed signal, $\hat{x}_k$, is a measure of similarity of the original signal $\mathbf{x}$ and a sinusoid with frequency $\frac{k}{n}$.
For the purposes of this assignment it suffices to notice that eq. (25) can be written in matrix form as

$$\hat{\mathbf{x}} = \Phi^* \mathbf{x} \qquad \mathbf{x} = \Phi \hat{\mathbf{x}} \tag{26}$$

with

$$\Phi = \frac{1}{\sqrt{n}} \begin{bmatrix} \omega_n^0 & \omega_n^0 & \omega_n^0 & \dots & \omega_n^0 \\ \omega_n^1 & \omega_n^2 & \omega_n^3 & \dots & \omega_n^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \omega_n^{(n-2)} & \omega_n^{(n-2)\cdot 1} & \omega_n^{(n-2)\cdot 2} & \dots & \omega_n^{(n-2)\cdot(n-1)} \\ \omega_n^{(n-1)} & \omega_n^{(n-1)\cdot 1} & \omega_n^{(n-1)\cdot 2} & \dots & \omega_n^{(n-1)\cdot(n-1} \end{bmatrix},$$

and $\omega_n = e^{-\frac{2\pi i k u}{n}}$. The remarkable fact that is leveraged by many ML algorithms such as GNNs is that the eigenvalues of *all* circulant matrices are precisely the columns of Φ.

Question 3.4 (6 points)

(a) (3 points) As mentioned in the box, it turns out that *all* circulant matrices are jointly diagonalizable: they all have the same eigenvectors which are the columns of the DFT matrix Φ. The eigenvalues of a circulant matrix are the entries of $\hat{\theta}$: the DFT of θ.
Given this information, show, in one line, that eq. (24) holds.

Hint: think about how diagonalizable matrices can be decomposed once their eigenvalues and eigenvectors are known.

(b) (3 points) The property of two matrices being simultaneously diagonalizable implies another important property that is a fundamental result in linear algebra. Can you state that property that is a direct consequence of matrices being simultaneously diagonalizable, and what that means practically if the two matrices considered are circulant with one of the two being the circulant with filter $\tilde{\theta} = (0, 1, 0, \dots, 0)$?

Hint: think about CNNs.

3.3 Graph Attention Networks

In what follows, we introduce the concept of attention to Graph Neural Networks. We can express the Graph Convolution Layer from equation 18 as follows:

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \frac{1}{\sqrt{D_{ii} D_{ij}}} W^{(l)} h_j^{(l)} \right) \tag{27}$$

Notice this is the exact same operation as in eq. 18, we only changed the notation of the nodes. The now equation indexes over each node of the graph such that $h_i^{(l)} \in \mathbb{R}^d$ is a vector of activations for a node i in the l-th layer. $\mathcal{N}(i)$ defines the set of neighbors for that node i (including self-connections). Notice this notation shows more clearly the aggregation of neighbor nodes. For simplicity, from now on we will ignore the normalizing factor $\frac{1}{\sqrt{D_{ii}D_{ij}}}$ leading to the following equation:

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} W^{(l)} h_j^{(l)}\right) \tag{28}$$

Question 3.5 (3 points)

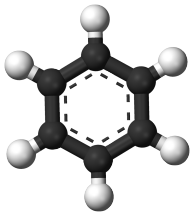
We want our Graph Convolutional Layer from equation 28 to attend over the neighbor nodes of node i. What would you add to the equation to perform attention? Provide the updated equation and justify your changes. If you get stuck, you can take a look at Graph Attention Networks paper.

3.4 Observing the Suitability of GNNs on Chemical Data

Unlike images or sequences, molecular constituent atoms do not have a canonical physical orientation or index-label ordering. The word canonical implies a natural computational representation. For example, the canonical representation of a sequence begins with the first element and ends with the last. The ambiguity in representing molecules is due to symmetries within the data and has deep implications in physics and chemistry. This can be seen with the benzene ring in Figure 3 where there is no obvious "first" or "last" atom and no particular expected orientation of the molecule. Here, we simply note that some molecular properties, such as energy in a vacuum, are invariant with respect to permutation of indices as well as invariant to rotation and translation in space. Graph Neural Networks allow us to encode these invariances into our neural representation.

In this problem, we will focus on the index ordering of the atomic constituents by creating an MLP and a GNN, which both attempt to predict the atomization energy of the molecule at 0K. There are a few things to note... First, the performance will increase significantly by standard scoring (z-scoring) the labels before training the network. Second, we will only use the atomic elements and bond type information to simplify the problem. This simplification means that classic techniques, like data augmentation with the MLP might perform well on this task. We discuss issues with MLPs, even with data augmentation, for chemical data within the questions.

Figure 3. A "ball-and-sick" model of a benzene ring. Notice that the carbon (black balls) and hydrogen atoms (white balls) are indistinguishable from other atoms of the same atomic number. Source: wikipedia.



This question involves implementing an MLP, and a GNN on a collection of molecular data called QM9 [5]. To help with the task of implementing a graph neural network, we will leverage the library PyTorch Geometric. Please install it into the correct environment using the **same version** of `cuda` (or `cpu`) as your PyTorch installation (see here for

an installation guide). PyTorch Geometric uses a sparse representation of edges and an unusual form of batching. Please consult Tutorial 7: Graph Neural Networks before proceeding.

Since it is an implementation question, you will need to fill in the missing pieces from the github repository for the class. There are three relevant files for this assignment `data.py`, `networks.py`, `train.py`. The file `data.py` is already complete; you will need to provide a solution only in the other files.

Your code will be checked by a unit test which determines if your implementation accurately accomplishes the spirit of the question. Each question must also have a short answer in Ans Delft.

Question 3.6 (20 points)

(a) (2 points) Take a closer look at the QM9 dataset that is downloaded automatically in the `data.py` file and examine the first few molecules. What's the problem if we wanted to train a Multilayer Perceptron to predict properties of this dataset, naively? How could we overcome it by leveraging our knowledge of the *entire* dataset (including test data)?

Hint: Consider if we found a new molecule and wanted to test our network on it.

(b) (7 points) Since we have access to the entire QM9 dataset, we can use our insight from the last question and implement a Multilayer Perceptron to predict the z-scored atomization energy at 0K, known in the QM9 data as U_0^{ATOM}.

Implement a Multilayer Perceptron which predicts this quantity using the one-hot representation of the atomic number `data.x[:, :5]` and the one-hot representation of the bonds as edge attributes `data.edge_attr`. Train the MLP. Report the validation loss curve and the final average test loss.

(c) (3 points) U_0^{ATOM} is a permutation invariant quantity. That means, if we encoded the exact *same* molecule, but changed how we indexed the atoms, we should predict the *same* output z-scored atomization energy.

Is that what you observe in the Multilayer Perceptron? Permute the test set's atomic indices only. What is the average estimated test loss of your model?

Hint: We provide a function that permutes the indices of a batch of molecules. Please apply the function for this experiment.

(d) (8 points) Implement a GNN which predicts a z-scored U_0^{ATOM} using only the one-hot representation of the atomic number `data.x[:, :5]` and the one-hot representation of the bonds as edge attributes `data.edge_attr`.

We recommend considering the graph convolutions RGCNConv, which considers categorical edge attributes, and / or MFConv, which was designed with molecules in mind. The node features may benefit from an embedding using a linear layer. After a global pooling operation, more layers may be applied, if necessary.

Report the validation loss curve during training. Next, report the test loss, then report the test loss again for molecules with permuted atomic index. What do you notice compared to the MLP? Consider hypothetically testing the GNN on a new molecule that isn't a part of QM9. what does the GNN input representation allow that an MLP does not?

Hint: Which pooling operation makes the most sense for molecules of different sizes?

References

[1] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 2

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 2

[3] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017. 2

[4] Christopher Olah. Understanding lstm networks, 2015. 4, 5

[5] Raghunathan Ramakrishnan, Pavlo O Dral, Matthias Rupp, and O Anatole von Lilienfeld. Quantum chemistry structures and properties of 134 kilo molecules. *Scientific Data*, 1, 2014. 13