Lecture 11 Script

11.1 Machine Learning on Sequences

Slide 1: Machine Learning on Sequences - Title Page

1. Graph Neural Networks extract information from data encoded on graphs. They are able to exploit underlying regularities in the data structure to create architectures that are stable, scalable and invariant to permutations.

2. Several applications, speech recognition and epidemic modelling being two examples, exhibit a time dependency in addition to the spatial structure encoded in the graph. The evolution of the process depends on the past values the process has taken up to this point. The time dependency as well as the variable lengths of data sequences motivate architectures specialized in dealing with sequential data.

Slide 2: Machine Learning on Sequences

1. A simple example to illustrate the estimation of properties of a sequence is to decide if a moving particle that we are not controlling is going to enter a forbidden area. In the figure we represent the forbidden region and several trajectories.

2. This trajectory stays clear of the danger zone.

3. And so does this trajectory.

4. But this one does not. We want to be able to tell this trajectory apart from the other two.

5. This is a problem that is time dependent. Suppose that we observe the trajectory at different points in time. Times 1 through 8 in the figure. It is clear that depending on the time index we have different degrees of confidence on the category to which the trajectory belongs.

6. At time 1 the three trajectories are the same.

7. They are still the same at time 2.

8. At time 3 they start to diverge and we could start making a determination that the red trajectory is on a collision path. But a determination is likely premature.

9. At time 4 we can have a larger degree of confidence.

10. And at time 5 we are but certain that the red trajectory in the middle will enter the forbidden region while the other two will stay clear of it.

11. At time 6 we are certain that the red trajectory is at fault while the other two seem safe.

12. At time 7 the two black trajectories up and down seem to be pulling away.

13. Something that we can ascertain with confidence at time 8.

14. This problem is not something that we can map to a sequence of classifications, which is something we would know how to do.

15. It maps to the classification of a sequence, which is not something that we know how to do. Indeed, the destiny of the trajectory is not a function of the current position only. It is a function of previous positions as well. Physical systems have inertia. The direction of movement is important.

16. An AI that maps the current position x_t to a prediction on the trajectory's class y_t

17. Is less accurate than an AI that maps the history of positions x_0-t to a prediction on the trajectory's class y_t.

18. We do not want to predict y_t from x_t.

19. We want to predict x_t from the trajectory's history x_0-t.

Slide 3: Unbounded Memory Growth

1. The challenge with making predictions on a sequence is memory growth. Predictions on a sequence of observations depend on the complete history of the process. As the iteration index t grows, the number of observations in which we base our predictions, also grows.

2. At time 1 we observe input x_1

3. And make prediction y_1-hat. This is the problem we have studied thus far. It can be easy or difficult. Depending on the dimensionality and structure of x_1.

4. At time 2 we observe input x_2.

5. We now have to make a prediction y_2-hat. But this prediction depends not only on the value we observe at time 2. It also depends on the observation at time 1. The amount of data we have to store doubles. More importantly, the complexity of the learning task squares. Remember that learning complexity grows exponentially with the number of input dimensions.

6. At time 3 we observe input x_3.

7. To make a prediction y-hat-3. This prediction has to depend on the observation x_3. But also on the observations x_2 and x_1. Memory has tripled. More importantly the complexity of the task cubes. Because the complexity of sampling a space grows exponentially with its dimensionality.

8. In general.

9. At any given time t we have a new observation x_t

10. And make a new prediction y-hat_t. This prediction is not only a function of the current state. But a function of the whole past. We have linear memory growth. With a consequent exponential growth in the complexity of the learning task. This growth is unbounded. Which is untenable.

11. Recurrent neural networks resort to the estimation of a hidden state to avoid this unbounded memory growth.

Slide 4: Markov Random Process

1. Before we talk about RNNs we need to talk about memory in stochastic processes. This will play an important role in motivating and understanding the architecture.The classical tool to study memory in time sequences is the Markov Random Process. We say a stochastic process is Markov, or memoryless, if the conditional probability of observing a

certain value at time t-plus-1 given that we know the complete history of the process from time 1 to t, is the same conditional probability of knowing only the state at time t.

2. More succinctly, the process is Markov if it is the same to condition on the current value x of t or to condition on the whole trajectory of the process.

3. This condition further implies that the future trajectory of the system, signified here by x_t-plus-one, is independent of the past, signified here by the trajectory values observed between times 1 and t-1. Provided that we know the present. Signified here by the state x_t.

4. This further implies that when it comes to predicting the future, knowledge of the past is irrelevant. Put differently, if we are interested in the future trajectory of the system, it suffices for us to know the value at the present point in time. If we are also given information about the past trajectory of the system, it doesn't alter our prediction of the future.

5. In addition to the state x_t, a Markov process may also have some outputs y_t. When this is the case, the outputs are assumed to be conditionally independent. The probability of the output taking some value y_t given only the current value of the stochastic process x_t is the same as the probability of y t conditioned on the entire trajectory of the stochastic process. An example of an output is the trajectory category. Whether we are entering the forbidden area or not.

Slide 5: Learning in a Markov Process

1. The reason for us to introduce memoryless Markov processes is that if we are given a Markov process, learning **is** equivalent to a sequence of learning problems. We do not have the challenge of learning from a sequence. We have the simpler challenge of a sequence of learning problems.

2. To see that this is true, note that the evolution of the state of the process, x of t, is a chain of memoryless transitions. That is, at every time step, the transition from x_t to x_t-plus-one depends only on the current value of the process. The past is irrelevant.

3. Moreover, the outputs of the process, y_t, depend only on the current state of the process. The output y_t depends only on the state x_t. The output y_t-plus-1 depends only on the state x_t-plus-1.

4. Thus, if we want to design an AI capable of predicting the output of the process, it is sufficient for the AI to learn how to mimic the conditional distribution of the observations y_t given the present state x_t. Since the past is irrelevant in nature, it is irrelevant for the AI. This is all good, but we have already seen an example where predicting the future of the trajectory benefits from information about the past. The reason why this happens is because the process in the example is **not Markov.** When processes are not Markov, we have to resort to architectures that take the history of the sequence into consideration. Onse suh architecture is the recurrent neural network that we introduce in the next video.

11.2 Recurrent Neural Networks

Slide 6: Recurrent Neural Networks - Title Page

1. Recurrent Neural Networks are the information processing architectures that we use to learn in processes that are not Markov. Namely, in processes where knowing the history of the process helps in learning.

Slide 7: All Processes are Markov. However Disguised

1. We reconsider here the problem of predicting whether a particle will enter a forbidden region. We have already seen that when predicting the future of the trajectory it is beneficial to know the whole past. We have also seen that the reason why this happens is because this process is **not Markov.** In turn, this is true because of physical inertia. Which is just a way of saying that the direction and speed of movement matter if we are trying to predict the future path.

2. If you need convincing, here is an example of a trajectory that enters into the forbidden area.

3. At time 4, the positions x_4 observed for this new trajectory and for one of our old trajectories are the same. But one of the trajectories is on a collision course with the forbidden area and the other one is on a safe trajectory. We can see that this is true (without looking at their futures) if we compare their positions at time 3.

4. (Empty)

5. The important observation for us to make now is that the process is not Markov as described. But if we also have access to the velocities and accelerations of the particle, the system becomes Markov. Having access to those unobserved states would allow us to predict the next position of the particle based on the current state of the particle. Now made up not only of the position x_t, but also including the velocity v_t and the acceleration a_t.

6. By the way, this depends a little on the model of your particle. It could be that you have to go deeper into the position derivatives to have a Markov model. You may need to have access to the Jerk and the Snap. Or even access to the Crackle and the Pop. But in any event, the important point is that all systems are Markov. This is true in a shallow intuitive sense and in a deep philosophical sense. The problem, however, is that we often lack enough information to observe their Markov structure. In this problem, we are observing positions. Because this is what we can measure directly. But we don't know the velocity or the acceleration. Not to mention the higher order derivatives if we need to go that route. The system is Markov. But on a state that remains hidden to the observer.

Slide 8: Hidden Markov Model

1. This fact brings us to the introduction of hidden Markov models.

2. We say that a stochastic process x_t follows a hidden Markov model if there exists another, unobservable stochastic process z_t that is Markov. And, furthermore, the state z_t completely determines the probability distribution of the observed state x_t.

3. Formally, the process z_t is one whose probability distribution at time t-plus-1 given the entire trajectory of the process is the same as its probability distribution when conditioning only on the current value of the process. This just means that z_t is a regular Markov process.

4. The new part of the definition.

5. Is to add the condition that the probability distribution of the observed state x_t given the current value of the hidden state z_t, is equal to the probability distribution of x_t given the entire history of the hidden process.

6. Thus, we have that the hidden state, z_t, is a memoryless Markov stochastic process

7. And that the observed state x_t is conditionally independent. It depends only on the current value of the hidden state z_t. Not on the entire **past** trajectory of the hidden state.

8. As before, we may have outputs of the process y_t. These are conditionally independent when given the **hidden** state z_t, Very importantly, this does not mean that they are conditionally independent when given the current **observed** state x_t. This is, ultimately, the reason why we can't have a sequence of learning problems. And we instead end up with the problem of learning from a sequence.

Slide 9: Machine Learning on Hidden Markov Models

1. Indeed. In hidden Markov models, learning is not equivalent to a sequence of learning problems

2. The hidden state z_t is a Markov process. It's probability distribution is a chain of conditional probabilities in which the state at time t+1 depends only on the value of the process at time t.

3. Given the state z_t of the hidden process, the probability distribution of the observation y_t is completely determined. We don't need the history of the process if we know z_t.

4. The probability distribution of the observable state x_t is also completely determined if we observed z_t. But this fact is not very important in this discussion. What matters most to us is that the observation y_t is conditionally independent.

5. Because given that this is true, to predict the outputs of thel stochastic process, it is enough for the AI to try to mimic the conditional distribution of y_t given the hidden state z_t. The AI could try to learn a mapping Phi from z_t, to the predicted output y-hat_t.

6. Implementing this map, however, requires access to the state z_t.

7. But this state is hidden. It is unobservable. We do not know the value that z_t has taken. This is the reason why the process is **hidden** Markov. Not **plain** Markov. What we know is the observable state x_t. But this is not sufficient for us to neglect the history of the process.

8. Recurrent Neural Networks are information processing architectures to bypass this issue. They utilize observations of the observable state x_t to estimate the hidden state z_t. And from this hidden state estimate, they estimate the process's output y_t.

Slide 10: Recurrent Neural Networks

1. In order to extract information from the data sequence without running into dimensionality issues, a recurrent neural network makes use of two separate learning parameterizations.

2. A function Phi_1.

3. That maps the observed state x_t.

4. And the hidden state z_t-minus-1.

5. Into a hidden state update z_t. Thus, at every time instant, the RNN updates its internal hidden state based on the received observation x t, and on the previous value of the hidden state, z_t-minus-1. Observe that we are using z_t to denote the internal state of the RNN. This state is not the same as the internal state of the hidden Markov process. It would be more accurate to use z-hat_t to denote the internal state of the RNN. But this complicates notation unnecessarily.

6. The second component of the RNN is the learning parameterization Phi_2.

7. Which maps the updated hidden state z_t

8. To a predicted output y-hat_t

9. This architecture receives the name **recurrent** because the hidden states are fed back as inputs for the next time step. This recurrence of the hidden state allows the RNN to encode past information it received from the data points seen so far in a manner that circumvents memory growth. By repeatedly updating the hidden state with each new data sample, the RNN creates an implicit mapping from the history of the process to the current hidden state. Without having to store and process all the samples it has seen so far.

Slide 11: Hidden State Update AI

1. So far we haven't said anything about the specifics of the parameterizations Phi_1 and Phi_2. They could be anything, but in a recurrent **neural network,** they are neural networks. More concretely, the AI for the hidden state update is a perceptron.

2. The state x_t is multiplied by a matrix A. The state z_t-minus-1 is multiplied by a matrix B. The results are added up and the sum is passed through a pointwise nonlinearity.

3. The learnable parameters of an RNN are the entries of the matrices representing the linear combination of the data points, A, and of the hidden state, B. Note that the number of learnable parameters does not depend on the time index t. That regularity prevents the number of learnable parameters from growing too large. And is also allows execution in sequences with variable length. Which is another important property that RNNs have.

Slide 12: Output prediction AI

1. For the output prediction AI Phi_2 we can use another perceptron.

2. We multiply the hidden state z_t by a matrix C. And pass the output through a pointwise nonlinearity.

3. It is also possible to use a multi-layer neural network for the output AI. The theme of this course is the exploitation of structure. RNNs exploit the structure of the sequence. On top of that, the observable state x_t, the hidden state z_t, and the observations y_t can also have some structure that we can exploit. If they are graph signals, we can use graph filters in lieu of the arbitrary matrices A, B, and C. Introducing this extra structure, leads to the introduction of **graph** recurrent neural networks.

11.3 Time Gating

Slide 13: Time Gating - Title Page

1. Recurrent neural networks are trained via backpropagation through time. But propagating the gradients over many states --- as necessary to train recurrent neural networks --- can lead to vanishing or exploding gradients.

2. Gated Recurrent Neural Networks address that issue by introducing gating mechanisms that create paths through time over which gradients do not vanish neither explode. Here, in particular, we will look into gating mechanisms in the form of long short term memory (LSTM) and gated recurrent units (GRU).

Slide 14: Vanishing / Exploding Gradients for Long Term Dependencies

1. In some learning tasks, the recurrent neural network may have to learn how to model long term dependencies of length T in the data sequence

2. That poses a challenge, however. Recurrent neural networks are trained via backpropagation through time, which relies on computing gradients with respect to the weights of the neural network and propagating them back in time. But the Jacobian of the hidden state z of t with respect to the corresponding weight matrix B will then depend on a chain of multiplications by that weight matrix. That is, initially, it depends on B

3. But when computing the next update, the Jacobian now depends on B squared

4. Then B to the third power

5. And so on, up to the length of the sequence, T

6. What happens is that, if the eigenvalues of the weight matrix B are small, raising them to the power of T will cause the gradient to vanish, thus leading to exponentially smaller weights

7. If the eigenvalues of B are much larger than one, however, raising them to the power of T causes the gradient to explode. Thus leading to exponentially larger weights B

Slide 15: Vanishing / Exploding Gradients for Long Term Dependencies (Example)

1. To see that, consider a simplified version of the RNN where we omit the pointwise nonlinearity sigma and the inputs x of t. That is, in this simplified model the hidden state is updated simply by multiplying its current value, z t minus 1, by the weight matrix B

2. At time T, the hidden state z T then depends on the T th power of the matrix B. That is, z T is given by the product between B to the T th power and z of t minus T

3. If the matrix B can be decomposed into its eigenvalues and eigenvectors as B equals to Q Lambda Q transpose, with Lambda a diagonal matrix made up by the eigenvalues of B and Q an orthogonal matrix made up by the eigenvectors of B, then the recurrence of the hidden state can be written as z T being equal to Q times Lambda to the T th power, times Q transpose, times z t minus T

4. Writing the recursion in this form allows us to see that eigenvalues less than one will vanish, while those eigenvalues that are greater than one will explode

5. Implying that any component of z t minus T that is not aligned with the largest eigenvalue will then be discarded

Slide 16: Gating Mechanism

1. To address this issue of vanishing gradients, we must add a gating mechanism to a recurrent neural network

2. Gates are scalars on the unit interval that act on the current input of the neural network, x t, and on the previous hidden state of the network, z t

3. Gates control how much of the input signal and past time information encoded in the hidden state should be taken into account at each time instant

4. Gates are updated at every step of the sequence

5. They are fundamental to address the issue of vanishing or exploding gradients because they allow the recurrent neural network to create paths through time that have derivatives that neither vanish nor explode

6. Through those gates, the RNN can then create dependency paths that allow encoding both short and long term dependencies of the data sequence

Slide 17: Long Short-Term Memory (LSTM)

1. The most popular gated RNN architecture is the long short-term memory model, which has been very successful in applications such as speech recognition and image captioning.

2. LSTM recurrent neural networks maintain blocks known as LSTM cells. Each of those cells has a self-loop, maintaining an internal memory, in addition to the overall recurrence of the recurrent neural network. The self-loop of a LSTM cell is defined in terms of three gates: a forget gate unit f t; an input gate unit g t; and a cell output gate q t [recall that gates are scalars on the unit interval].

3. Let then x t be the input of the LSTM cell; z t the hidden state of the overall recurrent neural network; and let s t the internal memory of the LSTM cell

4. That internal memory s t is updated by applying the forget gate, f t, to the previous value of the internal memory, s t minus 1, and by applying the input gate, g t, to the state update of the RNN, which, as we recall, is given by a nonlinearity sigma on top of a linear combination of the input x t and the previous value of the hidden state, z t minus 1

5. With the new value of the internal memory, s t, the output of the LSTM cell can then be updated by applying the cell output gate, q t, to a nonlinearity sigma applied to the updated value of the internal memory, s t

Slide 18: Gated Recurrent Unit (GRU)

1. Another popular model of gated recurrent neural network is the so-called gated recurrent unit, GRU

2. GRUs are a slight variation of the LSTM model, in that now a single gate u t plays the role of both input and forget gates. That is, the hidden state of the recurrent neural network is now updated as u t times the previous value of the hidden state, z t minus 1, plus 1 - u t times the original update of the hidden state, that is, the result of applying the nonlinearity sigma to a linear combination of the input x t and the previous hidden state z t minus 1

3. Note that in this the contribution of the previous hidden state, z t minus 1, to the updated state is controlled by the reset gate r t

4. Although those are arguably the two most popular architectures for gated RNNs, many more variants of gating mechanisms for RNNs exist.

Slide 19: Gate Computation

1. Note that, in long short-term memory cells and gated recurrent units, the gates themselves are calculated as the outputs of recurrent neural networks

2. For example, the forget gate of a LSTM cell, f t, has its own state variable z t prime that is updated by a recurrent neural network. That is, at each time instant, the internal state variable of the forget gate, z t prime, is computed by the combination of a pointwise nonlinearity and a linear combination of the input x t and the previous value of the internal state, z t minus one prime.

3. Given the updated internal state z t prime, the forget gate f t is then computed from the input x t and the internal state z t prime as a sigmoid nonlinearity applied to a linear combination of the input, x t, and the internal state, z t prime
4. Here, U and W are both linear layers mapping the input and state features to a single scalar
5. And the sigmoid activation function ensures that the gate values, f t, remain on the unit interval

11.4 Graph Recurrent Neural Networks

Slide 20: graph Recurrent Neural Networks – Title Page

1. We have introduced RNNs as architectures to learn features of time varying processes. We define now graph recurrent neural networks as particular cases in which the signals at each point in time are supported on a graph.

Slide 21: From RNNs to GRNNs

1. To be more precise consider a time varying process x_t in which each of the signals observed at each point in time is supported in a common graph S. In the figure we show three instances of graph signals observed at times t-minus-2, t-minus-1, and t. The figures are variation diagrams where the edges represent changes in signal values. The graph that supports the signals is the same at all times.
2. A graph recurrent neural network combines
3. A graph neural network because the signals x_t is supported on a graph.
4. And a recurrent neural network because x_t is a sequence

Slide 22: A Recurrent Neural Network for Graph Signals

1. To define a **G**RNN we begin by recalling the definition of an RNN. The component of an RNN that is different from usual neural networks, is a hidden state z_t that is updated according to the perceptron sigma of A x_t plus B z_t. We present here a block diagram that is more modular than the one we introduced earlier.

2. In this diagram the observable state x_t is fed to a linear block where it is multiplied by the matrix A.

3. The hidden state z_t is fed to a separate linear block where it is multiplied by the matrix B.

4. The outputs of these two blocks are summed.

5. And processed with a pointwise nonlinearity sigma to produce the state update z_t.

6. This update is fed back as an input to the linear block, where it will be processed in the next iteration to compute an updated hidden state z_t-minus-one.

7. The RNN involves a second perceptron. This one processing the hidden state z_t to produce the output estimate y-hat-t.

8. This perceptron composes a multiplication of the hidden state z_t with a matrix C.

9. With a pointwise nonlinearity sigma.

10. In this video we are interested in situations where the observed state x_t and the output y_t that we are trying to estimate are graph signals supported on a common shift operator S.

11. We are therefore going to required that hidden state z_t also be a graph signal supported on the same graph shift operator S. This requirement is not necessary. But as it is easy to foresee, requiring the hidden state z_t to be a graph signal allows for the use of graph filters and. This is likely to lead to architectures that are permutation equivariant and where we retain the stability and transferability properties of graph filters and conventional GNNs.

Slide 23: graph recurrent Neural Networks

1. To complete the definition of a GRNN we therefore require that the linear operations defined by the matrices A, B, and C be graph filters. We start by specifying the update of

the hidden state as one in which the hidden state and the observed state are propagated through graph filters.

2. Then, the matrix A is parametrically defined in terms of the shift operator S. And is furthermore given by the familiar polynomial form. The coefficients of the filter are denoted as a-sub-k. This is the filter that we use to process the current observed state x_t.

3. The matrix B is defined analogously. It is parametric on the shift operator. More concretely, it is a polynomial on the shift operator. The coefficients of which are denoted as b-sub-k.

4. The outputs of these two filters are added up.

5. And the result is processed with a pointwise nonlinearity sigma. This produces the hidden state update z_t.

6. The updated hidden state is fed back to become an input to the graph filter with coefficients a_k in the next iteration. Observe that in this architecture the blocks are all the same blocks that appear in the corresponding part of an RNN. The only difference is the use of graph filters in the blocks where a general RNN utilizes generic linear transformations.

7. For future reference we write the state update as the composition of a pointwise nonlinearity sigma with the addition of the graph filter A-of-S applied to the observed state x_t and the filter B-of-S applied to the previous hidden state z_t.

8. To estimate the output y_t the hidden state z_t is propagated through a graph filter as well. That is the generic linear transformation C is required to be a graph filter. This is a familiar polynomial on the shift operator S modulated with coefficient that we denote with c_k.

9. Thus, to estimate the output y_k, we multiply the hidden state z_t with a graph filter.

10. And process the output with a pointwise nonlinearity. The output is our estimate y-hat-t.

11. For future reference we write the output prediction as the composition of a pointwise nonlinearity sigma with the graph filter C-of-S applied to the hidden state z_t. This is a graph perceptron with coefficients c_k applied to the hidden state z_k.

Slide 24: Multiple Feature GRNNs

1. A GRNN is then made up of a hidden state perceptron along with an output prediction perceptron. In our definitions we write all of the graph signals as single-feature signals, which are proposed with single-input-single-output filters.

2. Each of the filters that make up the GRNN cab be replaced by a MIMO filter. Doing so yields a GRNN with multiple features.

3. This means that we end up with a hidden state update in which the matrix graph filter signal capital-Z_t is the result of applying the pointwise nonlinearity sigma to the sum of two MIMO graph filters. One of this MIMO graph filters processes the observable state capital-X_t. This is a matrix graph signal. The other MIMO graph filter processes the hidden state capital-Z_t-minus-1. This is also a matrix graph signal. The respective filter coefficients are the matrices A_k and B_k. Observe that in this architecture the matrices B_k have to be square.

4. We also end up with a prediction in which we estimate a matrix graph signal as an output. This is the result of applying a pointwise nonlinearity to the output of a MIMO graph filter whose input is the hidden state Z_k and whose coefficients are the matrices C_k.

5. The main advantage that follows from the use of a MIMO graph filter is that the hidden state Z_t can have larger dimensionality compared to the dimensionality of the observed states. In the language of graph signals, the hidden state Z_t can have more features at each node than the observed state X_t.

11.5 Spatial Gating

Slide 1: Spatial Gating - Title Page

1. We extend time gating to GRNNs to handle the problem of vanishing and exploding gradients

2. We discuss long range graph dependencies and the issue of vanishing/exploding gradients. We then introduce spatial gating strategies -- namely node and edge gating -- to address it

Slide 2: Gating in GRNNs

1. Similarly to RNNs, GRNNs can also experience the issue of vanishing/exploding gradients when encoding long term dependencies of graph processes

2. In long term dependencies, gradients vanish when the eigenvalues of B(S), the state-to-state graph filter, are much smaller than one, which in turn makes the weights B(S) exponentially smaller. And they explode when the eigenvalues of B(S) are much larger than one, making the weights B(S) exponentially larger

3. To address this issue, we do the same we did for RNNs: add a gating mechanism to GRNNs. As we will see in this video, GRNNs admit three different types of gating. So we define the gating mechanism in terms of generic operators Q hat of t and Q check of t. Q hat of t is the input gate operator, which acts on the input. Q check of t is the forget gate operator, which acts on the previous state

4. The input gate operator controls the importance of the input X_t at time t. The forget gate operator controls how much to remember, or forget, from the previous state Z_{t-1}. Observe that neither operator changes the dimensions of the signals to which they are applied

Slide 3: Time-Gated GRNNs

1. The simplest type of gating in GRNNs is time gating. This is just an extension of the input and forget gates we discussed for RNNs
2. In the Time-Gated GRNN, the input and forget gate operators are expressed as follows. The input gate operator Q hat of t is parametrized by a scalar lowercase q hat of t. The forget gate operator Q check of t is parametrized by a scalar lowercase q check of t.
3. These scalars multiply the filtered output and the filtered state respectively. And they both take values in the 0 1 interval.
4. In time gating, a single scalar gate is applied to the whole graph signal, that is, the same gate value is applied to the signal components at all nodes. The input gate either attenuates the input, or lets it all pass in the computation of the next state. Similarly, the forget gate either attenuates the previous state, or lets it all pass in the computation of the next state.

Slide 4: Long Range Spatial Dependencies

1. Even if the eigenvalues of the state-to-state graph filter are well-behaved, certain spatial imbalances can cause gradients to vanish in space
2. That is, certain nodes or paths of the graph might get assigned more importance than others in long range exchanges important for the task at hand
3. Examples of graphs for which this can happen are graphs with some type of community structure where the nodes within a community are highly connected
4. As we know from the discussion about long term dependencies in video 2, the gradients of the state Z_t depend on successive products of the state-to-state graph filter B(S). But since B(S) is a polynomial in the graph shift operator S, the gradients of Z_t actually depend on successive products of S.
5. Let T denote the duration of the graph process and consider the T th power of the shift operator S. When T is large, the matrix entries associated with highly connected communities will get densely populated
6. In other words, the subgraph corresponding to the community might become a complete graph, where all nodes are connected to all nodes. This, in turn, overshadows the local structure of the community, making it harder to encode processes with long range dependencies that are local on the graph

Slide 5: Spatial Gating

1. The issue of vanishing gradients in space can be solved by taking the node and edge structure of the graph into account in the gating mechanism.
2. We refer to this as spatial gating.
3. There are two strategies for spatial gating. The first is node gating. In node gating, separate input and forget gates are applied to each node. For instance, consider the graph on the left with a signal defined on the nodes 2, 3, 4 and 9.
4. Node gates allow gating each component of the signal independently. The components on nodes 2 and 3 pass, while those on 4 and 9 are attenuated.
5. The second spatial gating strategy is edge gating. In edge gating, separate input and forget gates are applied to each edge. On the graph on the right, we do not consider graph signals defined on the edges
6. But the edge weights between 2 and 3, 7 and 8, 7 and 12, and 9 and 10 can be attenuated to limit node exchanges across these edges
7. By taking the node and edge structures of the graph into account, spatial gating strategies help encode long range spatial dependencies in graph processes

Slide 6: Node-Gated GRNNs

1. We move on to formally define node-gated GRNNs. In the node-gated GRNN, the input and forget gate operators are expressed as follows. The input gate operator Q hat of t is parametrized by a vector lowercase q hat of t. The forget gate operator Q check of t is parametrized by a vector lowercase q check of t.
2. They correspond to multiplications of the filtered input and the filtered state by diagonal matrices
3. Where the diagonals are the input gate vector q hat of t and the forget gate vector q check of t. Each component of these vectors takes values in the 0 1 interval
4. Which makes it so that a different scalar gate is applied to each nodal component of the signal.

Slide 7: Edge-Gated GRNNs

5. As for the edge-gated GRNN, the input and forget gate operators are expressed as follows. The input gate operator Q hat of t is parametrized by a matrix Q hat of t. The forget gate operator Q check of t is parametrized by a matrix Q check of t.
6. The gating operation corresponds to elementwise multiplication of the graph shift operator by the N by N gate matrices Q hat and Q check of t
7. Whose individual entries are values in the 0 1 interval
8. Therefore, there is a separate scalar gate for each edge. By scaling the edge weights, edge gates control the amount of information transmitted across edges in local exchanges.

Slide 8: Gate Computation

1. In all gating strategies, the parameters of the input and forget gate operators are computed as the outputs of GRNNs themselves.
2. This means that gated GRNNs actually consist of 3 GRNNs. The first is the one used to compute the main state Z_t. The other two are the GRNN used to compute the input gate state Z_t hat, in blue; and the GRNN used to compute the forget gate state Z_t check, in green.
3. The computation of the gating operators' parameters from the input gate state and the forget state takes different forms depending on the specific type of gating.
4. In time gating
5. The scalar input gate q hat of t is computed by applying a fully connected layer c hat to the input gate state, followed by a sigmoid. And the scalar forget gate q check of t is computed by applying a fully connected layer c check to the forget gate state, also

followed by a sigmoid. The sigmoid activation function is necessary to ensure that these gates take value in the 0 1 interval.

6. In the case of node gating
7. The vector input gate q hat of t is computed by applying a graph filter bank calligraphic C hat to the input gate state, followed by a sigmoid. And the vector forget gate q check of t is computed by applying a graph filter bank calligraphic C check to the forget gate state, also followed by a sigmoid. Here, too, the sigmoid ensures that the values of the gates at each node are between 0 and 1.
8. In the case of edge gating, the computation of the gates is a bit more involved.
9. A matrix C hat is applied to the input gate state to learn linear features. Then, for each edge i j, we isolate the features corresponding to the endnodes i and j by multiplying the feature vector by N-dimensional direct deltas centered at i and j respectively.
10. The feature vectors corresponding to nodes i and j are then concatenated and passed through a fully connected layer lowercase c hat, which maps them to a scalar. Finally, this scalar is normalized by application of the sigmoid, to produce the edge gate corresponding to the edge i j. Once this process is completed for all edges i j, the input gate matrix Q hat of t can be constructed by assigning the ij th edge gate to the ij th matrix entry. Entries that do not correspond to edges of the graph are set to zero. The forget gate matrix Q check of t is computed analogously.

11.6 Stability of GRNNs

Slide 1: Stability of GRNNs - Title Page

1. In this lecture we discuss the stability of GRNNs. In particular we show that as GRNNs can be seen as a time extension of traditional GNNs they inherit their stability.

Slide 2: Relative perturbation model

1. We start recalling the notion of relative perturbations modulo permutation discussed in previous lectures.

2. For two graph shift operators S and S tilda

3. we define the set of perturbation matrices module permutation as

4. As the set of matrices E that satisfy the expression indicated in equation (1)

5. Where calligraphic P is the set of al permutation matrices

6. We measure the distance between S and S tilda as the minimum of the norm of the matrices E that relate S and S tilda in the set calligraphic E

7. Then if S tilda is a permutation of the shift matrix S, their distance equals zero

Slide 3: Lipschitz filters

8. Now, we recall the definition of Integral Lipschitz filters

9. We say that a filter A of S is integral Lipschitz if there exists a positive constant such that

10. The spectral representation a of lambda satisfies

11. the condition indicated in (2).

12. Notice that integral Lipschitz filters also satisfy that the absolute value of lambda times a the derivative of a of lambda is bounded by a constant

13. This implies that the frequency response of the filter becomes flat for large values of lambda

Slide 4: Assumptions

14. We consider a GRNN with one input feature, one state feature and one output feature, described Z sub t and y hat sub t as indicated. Additionally, we consider the following assumptions

15. The filters A, B and C are Lipschitz integral with constants C sub A, C sub B and C sub C, and whose norm is unitary.

16. The nonlinearity functions sigma and rho

17. are Lipschtiz with Lipscthiz constant equal to one,

18. and they have zero as a fixed point

19. The initial hidden state is identically zero, this means z sub zero equal to zero. Additionally the norm of x sub t is lower or equal than the norm of x for all t

Slide 5: The GRNN theorem

20. Now we present formally the stability theorem for GRNNs

21. Let S and S tilda the graph shift operators of the graph and the perturbed graph

22. And let E be a matrix in the set of relative permutation perturbation matrices such that the distance between S and S tilde is lower or equal than epsilon

23. Let y sub t and y tilda sub t be the outputs of the GRNNs running on S and S tilda respectively

24. And satisfying the assumptions A1 to A3. Then,

25. The minimum norm between y sub t and the permuted version of y tilde sub t is bounded as indicated in (3)

26. Where capital C is the maximum of the Lipschtiz constants and delta indicates the difference between the eigenvectors of S and E

Slide 6: Discussion

27. Then, we have that GRNNs are stable to relative perturbations with constant C times one plus square root of N times delta times T squared plus 3 times T.

28. Notice that C could be set at a fixed value or it can be learned from the data through the filters A, B and C. This is, it is a design parameter

29. Notice also that the term one plus delta times square root of N is a property of the graph perturbation, and it cannot be controlled by design

30. The eigenvector misalignment delta, measures the degree of commutativity of in the product of the matrices S and E

31. It is worth remarking that the stability bound depends on polynomial form from T which is due to the recurrence relationship in the computation of x sub t

11.7 Application: Epidemics

Slide 1: Title Page

1. In this part of the lecture, we explore an application of GRNNs -- epidemic modeling -- and compare them with GNNs and RNNs.

Slide 2: Epidemic Modeling

1. The epidemic modeling problem consists of modeling the spread of an infectious disease over a network of friends as a graph process, and using samples from this graph process to predict the number of infected people at a certain point in time.

2. The underlying graph is a friendship network built from real data from a high school in France

3. To model the spread of the disease we use the Susceptible-Infected-Removed or SIR model. At any point in time, each individual in the network is either susceptible to the disease, infected by it, or removed once they have contracted the disease and recovered.

4. We will compare the ability of 3 architectures, a GRNN, a RNN, and a GNN, to predict who will be infected in 8 days.

Slide 3: Friendship Network

1. To build the friendship network, we use real data corresponding to friendships reported by a group of 134 students from a high school in Marseille.

2. Each node of the graph represents a student

3. And each edge represents a friendship. All edges are unweighted and symmetric.

4. Isolated nodes were removed to make the graph fully connected.

5. In order to model the propagation of the disease, we will assume that friends spend time with each other and thus, the disease is likely to spread from friend to friend.

Slide 4: Susceptible-Infectious-Removed (SIR) Disease Model

1. To model the spread of the disease, we use the SIR model. At time zero, we randomize who has the disease. Each student may have it with probability 0.05. That is, roughly 1 in 20 students will have the disease at day 0.
2. At any point in time, each person is either susceptible, infected, or removed. Their states are updated every day, and the transitions between states are according to the following rules
3. A susceptible student can get the disease from an infected friend with probability 0.3.
4. A student that gets the disease will be infectious for 4 days, after which they recover
5. Finally, a student that is removed cannot contract the disease again nor spread it

Slide 5: Problem Setup

1. To keep the school safe, we need to predict who will get infected in 8 days.
2. The input to our neural network is a graph process x_t corresponding to the SIR state of each student. As such, the ith component of the signal x_t, which is associated with the ith student, is given by 0 for a susceptible student, 1 for an infected student and 2 for a student that is removed.
3. We are only interested in tracking the infected students. Therefore, we want to predict a graph process y_t where the ith element of y_t, corresponding to the ith student, is a binary label with value 1 if the student is infected and 0 otherwise.
4. GIven x_t through x_t plus 7, we want to predict y_t plus 8 through y_t + 15. Since y_t of i is a binary label for each student, this problem reduces to a binary node classification problem.

Slide 6: Objective Function

1. In the context of node classification problems, accuracy is the usually the go-to performance metric. However, in this particular problem, it is not a good choice. When

trying to maximize accuracy --- usually by minimizing the cross entropy loss --- one only penalizes misclassifications, without distinguishing between true positives and true negatives

2. In epidemic tracking it is essential to capture all the infected people, thus true positives are more important than true negatives. Which is why we choose to maximize the F1 score

3. The F1 score is a harmonic mean of two other performance metrics --- precision and recall.

4. Precision measures the proportion of students that were correctly predicted as infected to all of those that were predict infected. As we can see in the chart, this corresponds to True Positives divided by Predicted Positives.

5. In other words, precision is the proportion of correct positive predictions. Having a high precision means that we have little false positives.

6. Recall measures the proportion of students that were correctly predicted as infected to all of those that are actually infected. As we can see in the chart, this corresponds to True Positives divided by All Actual Positives.

7. In other words, a high recall means the model is good at predicting infections, that is, it produces little false negatives. Having a high recall means that we are not missing many infections.

8. Therefore, the loss function we minimize is 1 minus the F1 score. This maximizes the F1 score, which provides a better trade-off between false positives and false negatives than the cross entropy loss.

Slide 7: Results

5. We present the results of epidemic tracking for a GRNN, a GNN and a RNN, all with roughly the same number of parameters.

6. In the GNN, the time instants, that is, the sequence dimension, becomes input features. As such, the number of parameters depends on the length of the process

7. In the RNN, the nodal components, that is, the graph dimension, becomes input features. As such, the number of parameters depends on the number of graph nodes

8. The figure shows the F1 score achieved on the test set by each architecture, averaged over 10 random realizations of the dataset. We see that the GRNN ouperforms both the GNN and the RNN. This is not surprising, as we know that GRNN exploits both the spatial and temporal structure of the data, unlike the GNN, which only takes the graph dimension into account; and unlike the RNN, which only takes the time dimension into account