AINT351 - Test revision Torbjorn

Date: 09-01-17

What is a decision tree

- A set of ordered rules for classifying data
- Each node addresses an input variable
- Leaves assign labels to the data

Types of decision trees

- Classification trees have leaves with discrete classes
- Regression trees have leaves with numerical values

Advantages of Decision trees

- Simple and intuitive
  - Unlike artificial neural networks
- Good computational performance
- Robust
- Possible to validate a model using statist

Disadvantages of Decision trees

- Greedy algorithms can get stuck in local optimality
- The simple decision structure does not represent all problems effectively
  - XOR, parity
  - Produce large trees
- Can produce overly complex trees that do not generalise
  - Over fitting

Decision tree learning algorithms

- Information gain
- Gini impurity and Variance reduction
- MARS
- Conditional inference trees

How decision trees are constructed

- The data set is split recursively into sub-sets based in a single input variable
- Until
  - All sub-set has the same class
  - **Splitting no longer improves prediction**
  - sub-sets are too small
- Top-down induction of decision trees (TDIDT)
  - Greedy

How are the sets split?

- Identify candidate splits
  - Loop through all possibilities
- Apply metric to the resulting sub-sets
- Choose the split that produces the best sub-sets

Identifying candidate splits

- Each decision tree node descries
  - A rule for dividing a data set
- All rules have the same form, consisting of:
  - A variable to be considered
  - A threshold value to compare data points to
- Threshold values between the values found in the data set
  - Do not improve classification within the given data set

Information gain

- Information gain is the change in **entropy**
- Entropy is based on the probability estimates
  - For this the algorithm uses frequencies
- The expected information gain is the difference in the sum of entropy across the sub sets

Gini impurity

- Uses the square of the probabilities of each class
- The Gini impurity reflects the sum of the probability of each target class (squared) within a sub-set
- The improvement is the weighted sum of the Gini impurities

Variance reduction

- For regression trees
- Data points do not have a discrete class but a continuous value
- Calculate the variance of the node before the split
- Compare with the sum of the variances in the new nodes

Split quality - Purity metrics

- Quality in a decision tree is related to the purity of a given set
  - Entropy and Gini impurity measure diversity in a set of discrete data
  - Variance measures diverstiy in a set of continuous data
- The purity is measured in terms of the probability of each class, present in the given set

Improvement

- Improvement is the difference in quality between the original sub-set and the joint quality of the two new sub-sets
- Gini gain
  - Improvement based on Gini impurity
- Information gain
  - Improvement based on entropy

Evaluating a Decision tree

- Error rate
  - The proportion of errors across all instances
- Resubstation error
  - Error rate on the training data
  - Tends to be optimistic due to overfitting
- Test set error
  - The performance on a previously unseen set of test data
- Hold out
  - Reserving some data, often 20% for testing
  - N-fold cross validation

Repeated Testing

- We need to use as much of our data as possible to maximise performance
- Repeated holdout
  - Run several evaluations
  - Error estimates are averaged to yield an overall error estimate

  - Randomly chose test sets to use all data
  - Sub-optimal due to potential test set overlap

N-fold cross validation

- Divide data randomly into N sets (folds) of equal size
- Leave each sub-set out during training and test on that sub-set
- Calculate the mean performance across N folds

K-fold CV variants

- The problem with not doing k-fold is the trade of between training and testing data
  - You want the max out both
  - Training for accuracy
  - testing for validation
- split all data in K bins
- run k separate learning expierements
  - In each one of those, you pick one of those K subsets as your testing set
  - The remaining K-1 bins are put together into the training set
- Run this K times
  - Average the results from the experiments
- Used all data for training and testing
- K is commonly set to 5 or 10 based on experience
- Stratification
  - Each sub-set has the same proportion of data from each class
- Leave one out CV
  - fold size 1
  - Best use of data and most expensive

The bootstrap

- Very good method for small data sets
- Creates a training set by sampling with replacement
- Samples a data set of `N` instances `N` times to create a new training set of `N` instances
- Test on all instances

Bootstrap properties

- Very pessimistic due to high probability of single instances not making it to the training set

  - On each draw an instance in a set of size `n` has 1 - 1/`n` probability of not being picked
  - Training data will only contain 63.2% of all the instances
- Combine with resubstitution error for realistic error estimate

Pruning

- Deciding on stopping criteria
  - Stopping criteria that are too specific produce small, under-fitted decision trees, e.g. local minima
  - Loose stopping criteria produce large, over-fitted trees
- Pruning overcomes this problem
  - Use loose stopping criteria or grow a full tree
  - Remove sub-branches that are not contributing significantly to prediction accuracy
  - It can also be desirable to trade accuracy for simplicity

Reinforcement learning

- Learning what to do when
- What
  - Actions
- When
  - State
- Feedback
  - Reward

Representing a problem

- Markov decsision process
- **STAR:**
- `S` - Set of states
- `T` - Transition function
- `A` - Set of actions
- `R` - Reward function

Probabilistic actions

- Given a state `s` and an action `a`
- The resulting state `s'` is a probability distribution over the states

Policies

- a function to tell you what action to take in any state you come across
- Always asks, what action to i take from this state to get the next best state
  - This will give you a sequence, or a plan, to get the best or optimal states
- Policy is a mapping from each state and action to the probability of taking action a when in state s
- Policy is a set of states resulting in goal state
- The agent's probability of choosing a given action in a given state
  - `n(s, a)`
- Commonly based on a value function and an action selection mechanism
- Greedy action selection
  - Always selects action with the highest value
- Softmax
  - Probabilistic
  - Highest probability to action with highest values
  - Non-zero probability to all other options

Delayed reward

- Don't know how your current Policy will effect your reward until you have it
- I play a long game of chess, i mess up right at the end
  - or i mess up right at the start but make good moves afterwards
- You don't know what it is
- You just go until the end and then you have to figure out what was good action and what was a bad action
- The rewards are the way to work this out
- Minor changes to the reward matter
- Delayed reward is utility
- Utility allows you to take the immediate reward, which could be good or bad, to get the long term good reward

Bellman equation

- pi* maximises our long-term expected reward
- expected value of the sum of the discounted rewards at time t given pi
  - meaning following pi
- its expect because its non-deterministic

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \; E\left[\sum_t \gamma^t R(s_t) \mid \pi \right]$$

- Utility of this policy is the expected set of states that i will see from that point on given that i follow policy
- The difference the utility of a policy at a state is what happens if we start running from that state
  - Given we're following a policy that starts in that state
- To note:
- The reward of entering a state is not the reward of that state
- reward gives us immediate feedback
- utility gives us reward as well as long term feedback

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid \pi, s_0 = s \right]$$

- the optimal policy for every state
- returns the action that maximises my expected utility
- with regard to the optimal policy

$$\pi^*(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} T(s, a, s') U(s')$$

(going backwards) - once we go to the new state prime] - we look at the utility of that state - then we look over all actions - which action gives us the highest value of that - like the pi* - Then we know the action for s' - gamma it - add the reward of that state - this is the **bellman's** equation

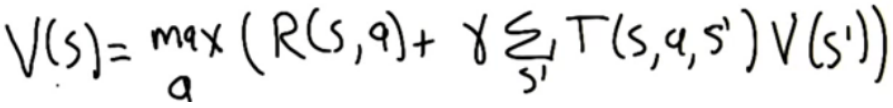
$$U(s) = R(s) + \gamma \max_{a} \sum_{s'} T(s, a, s') U(s')$$

- True utility of a state s
- is the reward of being of being in that state
- plus the discounted rewards of getting to that state
  - Which is defined as utility of getting there

Finding policies

- start by picking arbitrary utilities
- update utilities based on nearest neighbour
- repeat until convergence
  - one loop through is t
- have some estimate of utility at time t
- so use this to update utility to make them better using the eq
- update every iteration estimate of utility of state S
  - by recalculate it to be
  - the actual reward of state
  - plus the discounted utility that expect given the original estimates
- update all other states including self and weight those based on probability of getting there
  - given that i took action which maximises expected utility
- all possible because of the propagation of the truth which is the reward of entering the state
  - this truth goes across all states that were able to reach it
- this is value iteration
- Getting rid of the max because we're in a policy
- This utility is defined by the current policy

- The value of a state
- Is the max over all the actions
- The reward you get for taking that action from that state
- Plus the discounted value of the state you end up in
- Weighted by the probability you get there



Value functions

- Functions of states
  - or state action pairs
- That estimate how good it is for the agent to be in a given state
  - or how good it is to perform a given action in a given state

- The notion of "how good" here is defined in terms of future rewards that can be expected
  - in terms of expected return
- The rewards the agent can expect to receive in the future depend on what actions it will take
- Value functions are defined with respect to particular policies

State-value functions

- How good is a policy
- Evaluated in term of expected reward
- The bellman equation for `Vn`
  - Expresses the relationship between the value of the state and its successor states
- taking the reward from each successor state
  - Adding them up with a discount reward to get the final expected return
  - Do this for every policy

Policy evaluation

- Computing the value function, `Vn`, for an arbitrary policy, `n`
- Start with an arbitrary initial approximation, `Vo`
- Repeatedly apply the Bellman equation as an *update rule*
- Full backup, in place
  - Replacing `Vk` with `Vk+1` immediately

Policy improvement

- Should we change our given policy?
- The value of always choosing action, a, in state, s
- The policy improvement th

Value iteration

- Policy iteration requires many 'sweeps' in order to ensure improvement
- A simple back up can combine the policy improvement and policy evaluation steps

Model-free algorithms

- Estimate value functions directly

- Do not represent the transition function
- Transition probabilities will be implicitly included in the value function estimates

Monte-carlo methods

- We don't know the optimal policy yet
- We can follow a policy until we get to a state where we don't know which action to take
- We can then use monte-carlo methodology
  - Which is basically taking lots of random samples to try figure out which is good and bad
- To simulate lots of next actions and states
  - Follow these down for a while
  - behaving randomly
  - Get the expected reward
- That gives use an estimate of being in this state and taking this action
- **And since we might take the same action multiple times to get to these states**
- We can average over all of these possibilites
- Get lots of estimates
- Which gives use an estimate of a Q value function
- Building an evaluation function by using monte-carlo sampling and simulation
- Once you have new estimate from this state
- You can back that estimation up throughout all the previous known states
- And once this backed up you can now do selection again
- Do not assume that we know the transition function, `T`
- How can we evaluate a policy, `n`?
- Estimate based on *experience*
  - Obtain a set of *episodes* by following policy, `n`
  - Start at state, `s`, and take actions until the goal state is reached
- Use average discounted reward as an estimate of `Vn(s)`
- Estimate, `T`, based on *experience*

- Obtain a set of *episodes* by following policy, `pi`
- Use average discounted reward as an estimate of `Vpi(s)`

Monte-carlo policy evaluation algorithm

- Only consider first visit
- Initialise
  - Policy to be evaluated
  - An arbitrary state-value function
  - Returns - empty list for all states that belong to start state
- Repeat forever
  - Generate an episode using policy
  - For each state appearing in the episode
    - return following the first occurrence of s
    - Append reward to returns
    - Value of state = average of all returns

Monte-carlo properties

- useful for large state spaces as you don't have to find the underlying MDP by taking all states
  - Computational expensive
- You do need a lot of samples to get a good estimate
  - But if you have a large state space then this is ok
  - Most cases its not expensive to do
- Planning time is independent of number of state
  - this makes it good for scaling
- Running time is exponential in the horizon
  - Branching factor is action
  - so for every new action you find you have to find again
  - exponential amount of new actions because the tree grows

Estimating action values

- Without a model of the environment, `T`, we cannot chose optimal actions
- Estimate action values `Qn(s,a)`
  - Instead of state values `V(s)`
- Like state value estimation
  - Specific estimation for each state action pair

Exploring starts

- If policy is a deterministic policy
  - We may never visit some state-action pairs
- Guarantee that all first step of each episode gives all state-action pairs a non-zero probability
  - Ignore policy on first step

Temporal difference learning

- The difference in value estimates as we go from one time step to another
  - Value is a summary statistic for long term reward
- TD fall into value based learning
- Incremental way of bulding estimate
  - Look at the outcome of what you expierence through different episodes
  - and you use that to build estimate of values of various states are
- Has a learning rate
- Like monte-carlo
  - Learns from experience without a model
- Like dynamic programming
  - Estimation value estimates based on other existing value estimates
- Wait one step until the return following a visit to a state, `s` and an action `a` is known
  - What was the reward
  - What was the resulting state
- Use observations to update the value function
  - Don't wait until the goal state is reached
- TD resembles a Monte carlo method because it learns by sampling the environment according to some policy
- TD is related to dynamic programming techniques as it approximates its current estimate based on previously learned estimates
  - Bootstrapping
- 'Suppose you wish to predict the weather for saturday, and you have some model that predicts Saturday's weather, given the weather of each day in the week. In the standard case, you would wait until Saturday and then adjust all your models. However, when is Friday for example, you should have a pretty good idea of what the weather would be on Saturday - and thus be able to change, say Monday's model before Saturday arrives'
- Uses a running average
- Wait for one step to update value function

Advantages of TD prediction

- Do not need a model of the environment

- The bootstrap
  - Learn a guess from a guess
- The work in an on-line fashion
  - No need to wait for terminal state
  - Update every time step
- Do not have to ignore complete episodes with off-policy actions
  - More scope for exploring

TD control

- Must estimate state-action value function to take optimal actions
- Need exploring starts
- On policy TD control

Q-learning

- A greedy approach
  - Approximates the optimal policy
  - Independent of the policy being followed
- Requires continuous updates of all state action pairs
  - E greedy
- choose the best action possible
- get the reward
- update the table of with that reward
- choose another action
- repeat until goal

Episodes

- An episode is a traversal of the given environment
  - From a (random) start state to a terminal (goal) state

Trials

- A trial consists of multiple episodes
- The algorithm learns throughout a trial
- All trials use and update the same Q-function
- When do we stop?
  - Convergence
  - Minimum change in mean performance

Experiments

- Multiple trials
- The algorithm forgets what it has learnt between trials
- Provides mean and standard deviation of performance at each episode
- Learning reduces variation as well as mean performance

Methods for known transition functions

- Policy evaluation
  - Finds the value of a given policy
- Value iteration
  - Finds the optimal policy assuming greedy action selection

Model free methods

- Monte carlo
  - Taking the average of episodes performance
- Temporal difference
  - Learning from a guess
  - The difference in summary statistic for long term reward as we go from one step to another
- Approximate value functions from experience
- Do not model the transition function explicitly

Eligibility traces

- A record of the most recently experienced states, actions and rewards
  - A sequence of tuples (s, a, r) in chronological order
  - Updated on each step of the algorithm
- Our basic mechanism for *temporal credit assignment*
  - Spreading new information about state action values through the value function
- A bridge between MC and TD

Forward view

- Decide how to update the value of each state by looking at future rewards and states
- Theoretical
  - Not implementable as it is *acausal*

Backward view

- Mechanistic
  - Implementable
- Traces correspond closely to short-term memory
  - Our value function and transition function are long-term memory

Mixed updates

- MC method backed up values from the terminal (stopping state)
- TD/Q learning backed up values across one step
  - how one step effects the other and the difference between steps
- we can back up arbitrary many steps in between
- We can also back up many different steps
  - Must weigh them to maintain stats

Models

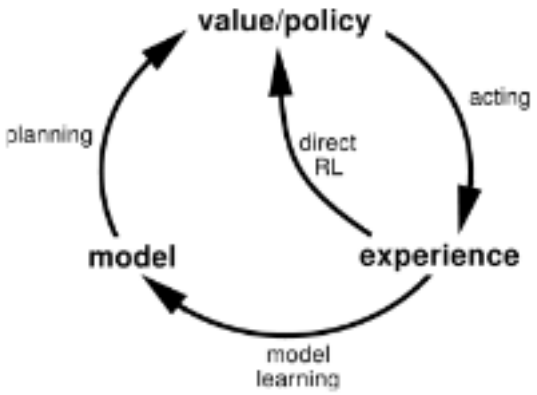
- Anything that can be used to predict results of actions
  - Simulated experience
- Distribution models
  - Provide probability distribution over successor states
- Sample models
  - Returns a single successor state drawn from the appropriate probability distribution

Planning

- Interpreted differently in different areas
- In RL, any process that:
  - Takes a model
  - Produces or improves a policy

Integrating learning, planning and acting

- Take actions
- Update value functions
- Update model
- Plan (simulate experience)
  - Update value functions



Dyna-Q

- Added on to Q-learning
  - Q-learning is model free - doesn't know transition matrix or reward
- Dyna does know
- Instead of taking an actual step in the real model
- We take a 'simulated' step
  - We can take many for every real step

Process: (including Q-learning)

- Init Q-table *iterate*
- Observe `s`
- Execute action `a`
- Observe `s'` and reward `r`
- Update q-table with new tuple `<s, a, s', r>`
- repeat

Dyna-q steps added on

- Learn models of `T` and `R`
  - Update model
  - `T'[s, a, s']` - The probability of being in state `s`, taking action `a` and ending up in new state `s'`
  - `R'[s, a]` - The expected reward if we are in state `s` and we take action `a`
- We simulate an experience
  - Randomly select `s`
  - Randomly select `a`
  - Infer `s'` from `T`
  - Get reward `r` from `R[s,a]`
- Update Q-table
  - Update using tuple from above steps

  - `<s, a, s', r>`
- Repeat last two tables many times
  - Cheap simulating rather than actually doing
  - use real step to simulate
  - once done many times can go back to real step
  - For every real step will be many simulated steps

RL so far

- Identifying optimal policies
  - T available
  - Dynamic programming
- Optimal policies from experience
  - Monte-carlo
  - Temporal difference
- Improving learning using modelling and planning
- Learning in complex and continuous domains

Table based solutions

- Everything is stored as a matrix or vector
- Transition function
- Policy
- State-value function
- State-action-value function (Q-values)

Table limitations

- Only works for small number of states and actions
  - Large amount of memory needed for large tables
  - Long time needed to fill table accurately
- Generalisation
  - Uses experience with limited subset of state space
  - Approximate much larger subset of state space
- Continuous values
  - Must always generalise

Discretization

- Taking a continuous value and making it discrete
- By hand
  - Arbitrary
- Automated

  - Unsupervised learning
  - Clustering criteria not necessarily conducive to learning

Function approximation

- Approximates a function from a subset of inputs and outputs
- We have a set of inputs and outputs and we should be able to get a function where were can get more outputs from inputs
  - We create the function
- Supervised learning
  - Linear regression
  - Regression trees
  - AI neural nets
- Combined with RL
  - State value function approximation
  - State-action value function approximation

Value prediction with function approximation

- `Vpi` is now a function in parametrised functional form
- The value function `Vt` depends totally on a vector of parameters is typically much smaller than the number of states
- Changing a parameter changes the value of many states

Backups

- An individual backup
  - of state
  - of state value
  - of value
- Temporal difference back up
- Each back up can be interpreted as an example of desired input-out behaviour of the estimated function
- with tables, the update to the function are relatively straightforward

Using neural networks for the value function

- Q value f multiple state action combinations
- Update function based on experience

Suitable function approximations methods

- Most classifiers assume a static training set
  - Multiple passes over training set is common
- In RL, learning must be *online*
  - Incrementally acquired data
  - Non-persistent data
- In RL we also typically have *nonstationary target functions*
  - Bootstrapping means target function is changing
  - Approximators must be able to handle this

Evaluating function approximators

- Many approximators minimises mean squared error (MSE)
  - Between the true value function `Vpi` and the estimated value function `Vt`
  - *Weighted by* `P(s)` *for a flexible importance*
- Not clear how MSE is related to learning performance
  - Best option available at the moment

Control with function approximation

- For control we will aprroximate the state-action value function
  - `Q(s, a)`
- A training example now becomes
  - `St, At -> Vt`

State space coding

- Assume state space that is continuous and two-dimensional
- One kind of feature could be circle in this state space
- Coarse coding
- Binary input vector

Radial basis functions

- RBFs are natural extension of coarse coding
- A collection of Gaussians
- The value of each feature varies with the distance from the mean of that feature
- Value between 0.0 and 1.0

Generalisation

- Over training the model
- Gets very good at working with the training data but when it comes to the testing new data it fails misserably
- unable to cope with new data is saying that the model is incapable of generalising
- Can make predictions of what has happened in the past about what will happen in the future
- Leverage learning in states where we have been to try to make predictions about states where we havent been

Partial observability

- Commonly we cannot uniquely identify the state of the world
  - Limited sensors
  - Sensor noise
  - historical events
- Need memory to tell states apart
- Replace states with observations

Limited sensors in grid world

- Encode what walls are present
  - Add up the present walls
  - North 1, East 2, South 4, west 8
- The observation is the addition of all walls that surround a state
  - If a state can only move north
  - then its observation is east = 2 + south = 4 + west = 8 = 14

Pole balancing

- MPD
  - State includes cart position
  - Pole angle
  - Cart speed
  - Rotational speed of pole
- POMDP
  - State includes only position and angle
  - Algorithm must implicitly estimate speeds based on memory of observed position and angle

POMDP Formalisation

- Underlying MDP
  - States
  - actions
  - transition function

Limited Observability

- Observations
- Observation function/model
- O[s', a, o] is the probability Pr(o|s', a)
- POMDP model
  - `M = <S, A, T, R, Observations, O>`

Objective and belief state

- Our objective remains the same
- As we cannot access the state, s, directly
- We introduce the belief state
- A belief state is a probability distribution

Belief update

- In a belief state b, we take an action a and make a new observation o
- What is the resulting belief state b'
- `b'[s'] = Pr(s'|o, a, b)`

POMDP value functions

- Continuous belief states
- We cannot make a table to keep up with continuous belief state values

Finite horizon POMDPs

- Claim
- For all the value times step t
- over all belief states b
- Can be written in a finite way
- The maximum over some set of finite vectors gamma sub t
- The maximum of the dot product of the vectors in that set with the belief state

  - Where dot product is the sum of over states and weighted probability of being in that state
- This function is called piecewise-linear and convex
- Because if you put all probabilities on a graph between one and zero
- the chances of being in that state are all linear lines
- Once you have all linear lines it looks like a cup (convex shape)
- If you want to find the probability that you're in that state
  - Along the x axis, you take the maximum value of the lines that cross that point
  - therefore you always take the surface line
- Have value functions that are piecewise-linear and convex
- Can be represented as the upper surface of a set of linear segments

POMDP Value iteration

- The value of belief state given that we're in time step t
- is equal to the maximum over all actions
- The reward that we get from being in a belief state and taking action
- Plus the discounted expected value of where we end up
- The reward is the average award given if you were in a belief state
  - Can do dot product of belief state plus probability distribution of rewards from possible states
- operations on bags of vectors and therefore is finite
- Finding all vectors
- Construct all policy trees using DP approach
  - Each action and observation adds new vectors

Naive algorithm complexity

- The size of set of all t-step policy trees grows doubly exponentially with time
- This becomes overwhelmingly large even for moderately sized problems

Reducing the number of a-vectors

- Only add trees that contribut to the maximum (witness)
  - For a sub tree, can we reach an optimal belief state
  - if not, dont add more vectors
- Remove vectors that are dominated (incremental pruning)

  - Never contribute to the maximum

Point-based value iteration

- Maintaining a full set of vectors is too complex
- Point-based algorithms approximates `Vt(b)`
  - Keeps only a sample set of vectors

POMDP and memory

- Constructivist approach
  - Do not require access to state space or transition functions
- Long term memory
  - Previously seen sequences of observations, actions and rewards
  - previous episodes
- Short term memory
  - Recently seen sequence of observations, actions and rewards
- state identification
  - Implicit
  - previously seen sequences that best match recently seen sequences
  - best match between LTM and STM
  - One can assume that if it has seen the same set of observations and actions together than they're in the same sequence of steps
  - If that sequence leads to reward then it is good

instance based methods

- Do not construct a theory
- Does classification/regression from the raw data each time
- Complexity is limited by the size of the training set

K-nearest neighbour learning

- Select the k points in the training set that are nearest to the data point you need to classify
- Requires a distance metric for deciding how close points are
- Euclidian distance
  - Consider each data point variable as an individual dimension
  - Normalise to avoid some dimensions dominating
- Class is the majority class of the k-nearest neighbours

Instance based solutions

- Nearest sequence memory
  - Keep current episode in STM
  - Remember last N episodes (instances) in LTM
  - Match STM and LTM (k-nearest neighbours)
  - Calculate highest value action
- Can learn very quickly
  - Potentially one shot
- Fixed memory requirements
  - Can forget solutions

Compared to belief state solutions

- Belief state LTM
  - Transition function
  - Observation function
- STM
  - Belief state
- NSM belief state is implicit in the K-nearest sequences