UNIVERSITY OF OSLO

Faculty of Mathematics and Natural Sciences

IN3050/4050 — Introduction to Artificial Intelligence and Machine Learning
Exam Spring 2020
Exam content: Around 2 working days
Duration: From May 25 at 2:30 PM to June 2 at 2:30 PM
Permitted materials: All

- **General information about exams the spring 2020 can be found here.**
- **You should deliver your exam answers as a single PDF file through Inspera. Info on how to deliver files in Inspera can be found here.**
- **If you become ill and cannot deliver your written exam within the deadline, see here for procedures. There are given no postponements. There will be arranged a resit exam in August for they who get ill.**
- **You are free to make plots/figures in any program you want, and add them to your delivered PDF. That includes using drawing tools on the computer, and drawing figures by hand and taking a picture of them. Some tips are available here.**
- **Your delivery should be anonymous. Do not write your name.**
- **If you have questions related to this exam, please submit them here.**
- **Please check the "Messages" on the Course Website daily. We will post any clarifications around the content of this exam there.**
- **You may answer in English or Norwegian.**
- **You are not required to write any code for any of these tasks – all calculations may be done by hand, and we do want you to show each step of your calculation.**
- **The examination answers must be the result of the student's own efforts. It is okay to discuss theory and assignment text with others. It is also okay to get hints on how a task can be solved, but this should be used as a basis for your own answer and not be copied unchanged. Sharing code or (parts of) the solution of a task is not allowed. If you include text, program code, illustrations or other items from the internet or elsewhere, you must clearly mark it and indicate the sources – however, in an independent assignment this is something that rarely happens.**
- **The tasks of the exam are given points summing to 100, giving you an indication of how each task is weighted and roughly how much time you should spend on it.**

1) Search/Optimization (7p)

a) Draw a 1-dimensional search landscape (a continuous landscape f(x) over a single variable x). (1p)

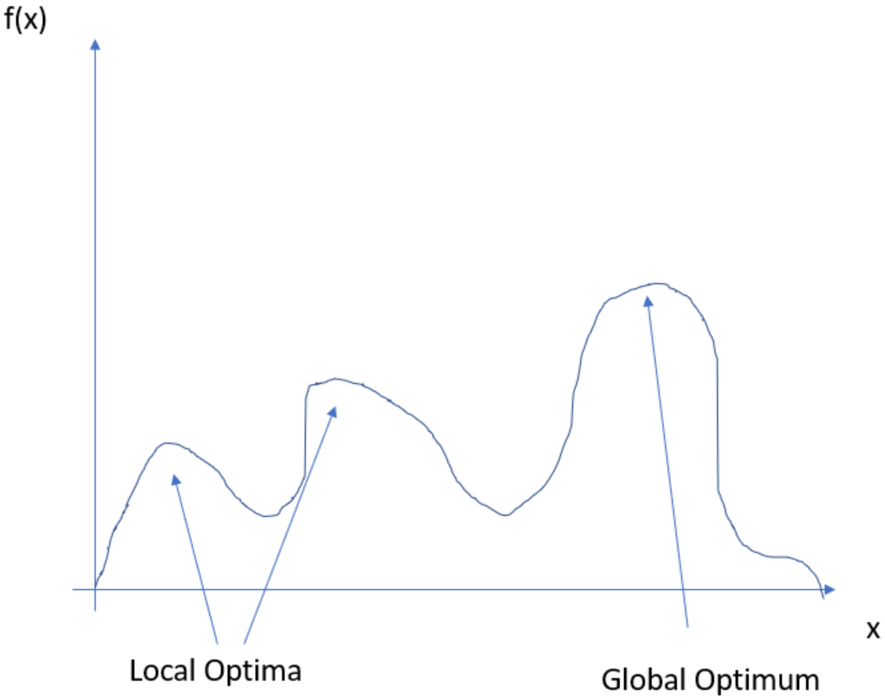
Use it to explain the concepts (1p for each)

b) Local optima
c) The global optimum
d) Exploration
e) Exploitation

f) Is a pure-exploitation algorithm likely to find a local optimum? How about the global optimum? (2p)

Suggested solution:

a) 1 point for a sensible search landscape with a few global/local optima, as for instance the one below.



b) Local optima: Those solutions that are better than others near them.(1p)
c) Global optimum: The solution that is better than all other potential solutions. (1p)
d) Exploration: Moving (randomly) around in the search landscape, trying to find new types of solutions that potentially are better than what we have found before. (1p)
e) Exploitation: Slightly changing solutions we already know are promising, to hopefully make them better. (1p)
f) Yes, pure exploitation can find the local optimum by gradually improving solutions until we can't find a similar, higher-performing solution. (1p) But it is not likely to find the global optimum for a complex problem, since complex problems typically have very many local optima, and we need to explore many of them for it to be likely to find the best one. (1p)

2) Evolutionary Algorithm: The Social Distancing Game (14p)

The World Health Organization (WHO) have approached you with a task to help people maintain a safe distance from each other during virus outbreaks. As a simplifying assumption, assume we are dealing with optimizing the intra-person distance in a square room (NxN meters) where each 1x1 meter cell can hold one person at most (the WHO has stressed that any solution with more than 1 person in a cell is *invalid* and they don't want the EA to even consider such solutions). The room should contain exactly N people. Below is an example of a 4x4 meter room holding 4 people (indicated by X's).

X			X
	X		
		X	

a) Design an Evolutionary Algorithm capable of optimizing the social distancing in square rooms of NxN meters, with the objective of maximizing the minimal distance between any two persons in the room. Decide on and briefly justify your choice of the parameters below. (Be brief, but specific. The full answer to this should not exceed one A4 page – up to 2-3 lines should be more than enough to describe each point.):

-Genotypes. Show an example-genotype corresponding to the figure above.
-Phenotypes
-Fitness Function (write out the full calculation here, in mathematical formula or in code)
-Mutation Operator
-Crossover Operator
-Initialization Criterion
-Termination Criterion
-Selection Operator(s)

(0.75p for each)

b) Are there any issues you have to pay special attention to in order to ensure your mutation, crossover and initialization does not produce invalid solutions? (2p)

c) You notice the EA suffers from premature convergence. Explain why fitness sharing can help you avoid premature convergence and describe how you could implement fitness sharing for this problem. (3p)

d) You're wondering if hybridizing the search can help you reach well-performing solutions faster.

1) Suggest a way to hybridize this EA.
2) Do you see any conflict between the goal of avoiding premature convergence and hybridizing the EA?
3) Referring to the concepts of exploration and exploitation, how do the fitness sharing and hybridization affect the EA search?
(3p total)

Solution

a)

There is quite a lot of freedom in making your own choices here, but the key is to check if students have gotten an intuition on how to formalize a problem as an EA problem, and choosing appropriate representations and operators. Sensible and well-reasoned answers can get a full score even if they do not directly match the ideas below. 0.75p for each of the 8 EA-components below.

Genotype: Could either be a list of tuples (e.g. (4,4)) or a list of integers uniquely identifying each occupied cell. Other types may work, but importantly, the genotype should be able to represent all valid configurations. The example-genotype for the list-representation could be [1,7,12,13]. *Important: The genotype should be able to give a 1-to-1 mapping to a room-layout like the one above.*

Phenotype: This could be a drawing of a configuration like the one above, or another representation that *lets us calculate the fitness of a solution easily*. It could even be the same as the genotype, if the student specifies how to calculate the fitness directly from this genotype.

Fitness function: For instance, the distance between the two closest individuals: sqrt($(x-x')^2 + (y-y')^2$) where (x,y) and (x',y') are the two individuals with the smallest distance. Other distance measures (e.g. Manhattan distance) could also be used. Note that average distance is less informative. *Important: Here, the students need to show that fitness evaluation should generate a single number evaluating a full configuration. That is, it does not evaluate one or a pair of persons, but the whole room!*

Mutation: Many possibilities, but important to make sure that mutations remove one (or a few) persons from their spot, *and insert exactly the same amount of new persons. And, that these new persons are not in the same spot as anyone else. Also important to ensure mutation does not produce any impossible locations (e.g. (6,5) in a 4x4 grid).* Too disruptive mutations (e.g. completely shuffling all persons) could be destructive and should be avoided.

Crossover: Would be ok to not include crossover. If included: Could for instance be splitting genotypes at a random point and swapping tails. But *again important to ensure the new genotypes are conflict free (no overlapping persons).*

Initialization: *The key is to make many different, valid genotypes.* For instance randomly sampling tuples or integers, but with the *important constraints of 1) no two identical tuples/persons in the list, and 2) all tuples/integers inside the border of the grid.*

Termination: Many possibilities, e.g. *reaching a certain desired distance, or a certain number of generations, or when solutions stop improving.*

Selection: Any reasonable selection mechanism from EAs could be applied. E.g. fitness-proportionate, rank-based, tournament.

b)
Issues you have to pay special attention to: See the parts in italics under mutation, crossover and initialization. The key is to ensure all new solutions 1) have the right amount of people, 2) that no two of them are in the same spot, and that 3) no persons are inserted “outside” the NxN grid. (2p)

c)
Premature convergence means the EA starts *settling on a part of the fitness landscape with a given type of solutions too early*, when we prefer that it explores many types of solution. Fitness sharing can help avoid premature convergence since individuals are forced to *share a part of their selection probability with similar solutions* during selection. This makes sure that if many solutions occupy one spot in the search landscape, *individuals from other spots are prioritized even if they have lower fitness*, which reduces convergence. (1.5p)

The idea in fitness sharing is dividing the base fitness of a solution by a value *reflecting how many similar solutions there are*. To do so, we need a way to *measure similarity* between solutions.

Gaining points here requires a sensible way to *measure similarity between solutions*. There are many options. A simple but imprecise one could be to count the number of identical tuples

between two genotypes. A better, more precise one would include building a mapping between the people nearest to each other between the two solutions, and calculate the total distance between all such pairs. A simple solution to this should be enough for a full score here, as long as it gives a little bit higher similarity score to the most similar kinds of solutions, allowing us to potentially leave a local optimum.
(1.5p)

d) (1 point for each part below)

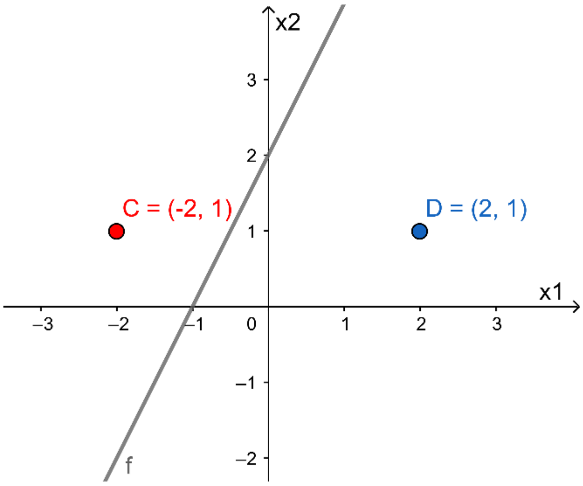
1) Hybrid EA: Means using some heuristic/other search method to improve individual solutions inside the EA. Any heuristic/search that improves individuals in *more structured ways than random mutation can be accepted here*. For instance, doing a *local search around each solution*. That is, making a small change (such as moving one person to a neighbour spot), measuring the fitness and keeping the best of the original and modified solution. The local search can either be long, searching through a long sequence of improved solutions, or stop as soon as it finds an improvement. However, it is important that information from the local search is inserted back into the EA population, either by inserting the best individual found (Lamarckian evolution) or the fitness of the best found solution (Baldwinian evolution). Discussion on Baldwinian/Lamarckian evolution are not required for a full score, I am just mentioning them here to indicate that both types of solutions are ok.

2) Conflict with avoiding premature convergence: Hybridization can reduce diversity by *optimizing many individuals in the same way, leading to premature convergence*.

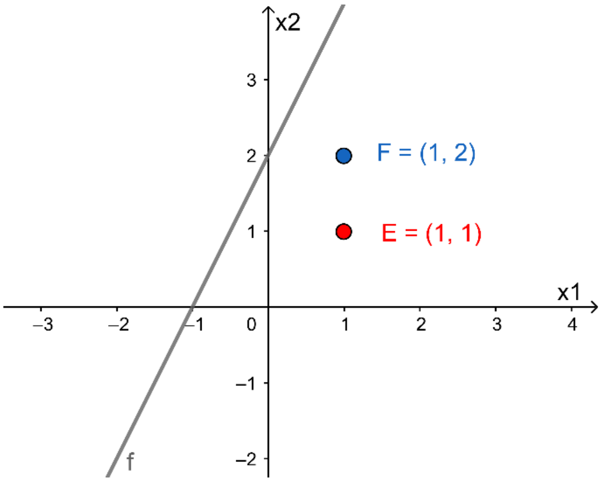
3) *Hybridization produces more exploitation, while fitness sharing produces more exploration.*

3) Classification (10 points)

We are considering a classification problem with two classes: The positive class, which we represent with the numerical value 1, and the negative class, 0. There are two features, x1 and x2, both are real numbers. We have trained a perceptron classifier on the training data. This has resulted in the decision boundary shown as line *f* in the figure. Points to the right of the line, e.g., the blue D = (2,1), are classified as 1, and points to the left of the line, like the red C = (-2, 1) are classified as 0.



a) Make an expression for how the classifier makes its decisions. Explain how you derived the expression. (Hint: Do not forget the bias). Show how this expression is used to classify C and D. (3p)



b) We pause during training, and we consider two observations from the training data: E = (1,1) which is wrongly classified, and, F = (1,2) which is correctly classified. Show how the perceptron training algorithm will update the weights when considering point E. Then show how it will update the weights when considering F after it has considered E. Make the necessary assumptions and state them clearly. (2p)

c) In a task like this, there are some parameters we have to set manually. One of them is the learning rate. There is no fixed value of the learning rate which fits all problems. Discuss with respect to the perceptron algorithm why we should make the learning rate not (too) big and not (too) small. (3p)

d) Suppose that you instead train a classifier using gradient descent for linear regression. Discuss why we should not make the learning rate (too) big nor (too) small in this case. (2p)

Solution

a)

The classifier has the form

f(x1, x2) = 1, if g(x1, x2) = w0 + w1*x1 + w2*x2 > 0

f(x1, x2) = 0, otherwise

for some choice of w0, w1, w2.

(assuming a bias of 1, alternatively follow Marsland: - w0 + w1*x1 + w2*x2 > 0)

(1 p)

The decision boundary has the form

g(x1, x2) = w0 + w1*x1 + w2*x2 = 0

We substitute in the two points where it intersects the axis

w0 + w1*(-1) = 0

w0 + w2*2 = 0

Hence: w0 = w1 = -2*w2

(1 p)

Remains to determine the sign.

Since g((0,0)) should be > 0, w0 > 0

One possible solution:

f(x1, x2) = 1 if g(x1, x2) = 2 + 2*x1 – x2 > 0
f(x1, x2) = 0, otherwise
(0.5p)

f(C) = 0, since g(C) =2 + 2 (-2) – 1 = -3
f(D) = 1, since g(D) 2 + 2 (2) – 1 = 5
(0.5p)

b)
We choose a learning rate *e* of 0.1.
(other learning rates are acceptable and will give different numbers below)

Adding bias, E yields the form E' = (1, 1, 1)
g(E') = 2 +2 – 1 = 3
f(E') = y_E' = 1
and the target is t_E = 0
Using the update rule from Marsland (formula 3.5)
wi = wi – *e*(y_E' – t_E):

w0 = 2 – 0.1 * (1 – 0) * 1 = 1.9
w1 = 2 – 0.1 * (1 – 0) * 1 = 1.9
w2 = -1 – 0.1 * (1 – 0) * 1 = - 1.1

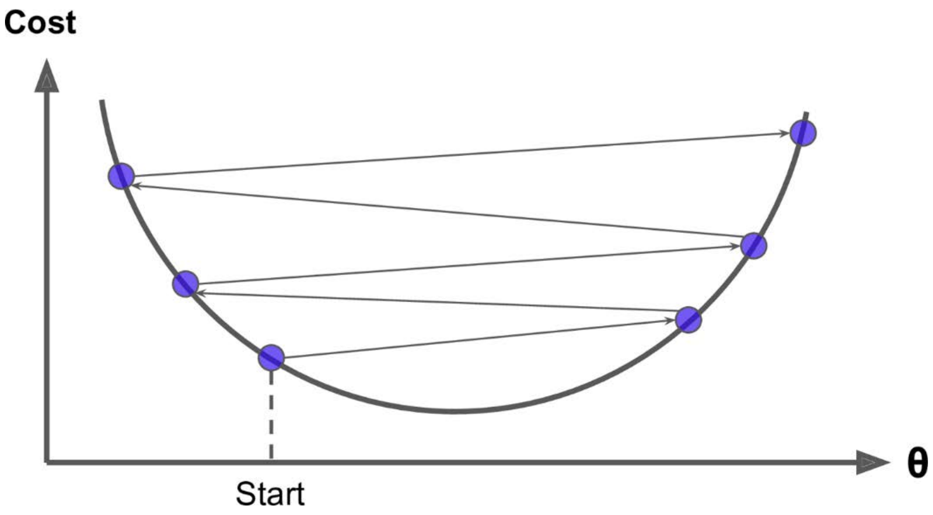
Adding bias, F gets the form F' = (1, 1, 2)
t_F = 1
g(F') = 1.9 + 1.9 – 1.1 * 2 = 1.6
f(F') = 1 = t_F
The weights are unchanged.

Comment: Students who have made mistakes in (a) and carry with them a wrong function, should still get a full score on this point if they follow the algorithm correctly

c)
Perceptron
Too large learning rate
Suppose a problem is linearly separable. Then we know that with a small enough learning rate, the training will converge and find a decision boundary which separates the two classes. If the learning rate is too large, it may miss the separating decision boundary and bounce back and forth. (2p)

Small learning rate
It takes longer to train with a small learning rate (1p)

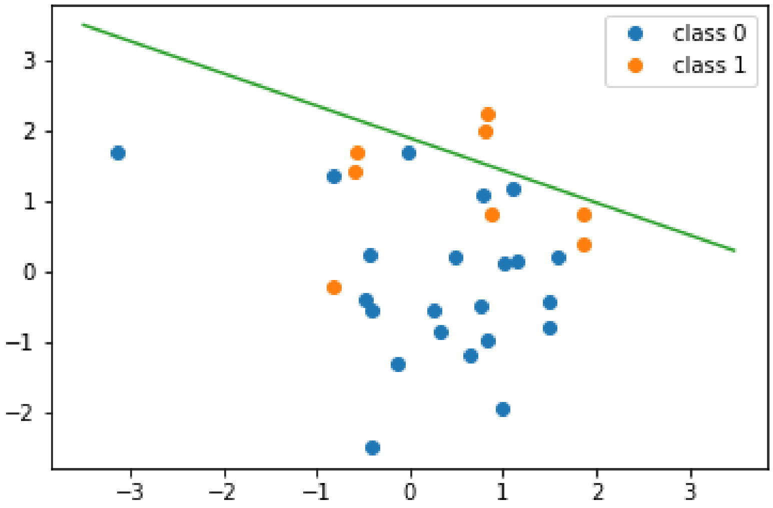
d)
Linear (and logistic) regression
The goal is to find the minimum of the loss function (error function).
This is a convex problem. By using gradient descent, we can come arbitrary close to the global minimum, if we use a sufficiently small learning rate. By using too large error rate, we may risk "going up the valley", see figure (from Gueron, *Hands-on Machine Learning*) (1.5p)



Small learning rate
Also with gradient descent, it takes longer to train with a smaller learning rate. (0.5p)

4) Evaluation (8 points)

We have trained a classifier and we are now ready to evaluate it on a test set of size 30. The result is as shown in the figure, where the green line indicates the decision boundary.



a) What is the accuracy of this classifier? Show how you found it. (2p)

b) How are the evaluation metrics precision and recall defined? What is the precision and recall of this classifier for each of the two classes? Show how you found them. (3p)

c) Formulate in your own words, why and under which conditions, we should use precision and recall, and not only rely on accuracy. Illustrate with examples of real-life problems. (3p)

Solution

a) Accuracy is the ratio of correctly classified items to the total number of items. We see that 6 items are wrongly classified, hence
Accuracy = (30 – 6)/30 = 0.8

b) To define precision we can use a confusion matrix

		True class	
		pos	Neg
Predicted class	Pos	tp	Fp
	neg	fn	tn

Where
Tp: True positives
FP: False positives
FN: False negatives
TN: True negatives
Precision = tp/ (tp + fp)
Recall = tp/ (tp + fn)

Class 1:
P = 22/28 = 11/14
R = 22/22 = 1

Class 0:
P = 2/2 = 1
R =2/8 = ¼

Comment:
The exercise does not say which class is on which side of the decision boundary.
(No penalty for not pointing this out and choosing the interpretation above.)
If one uses the other option, the results are

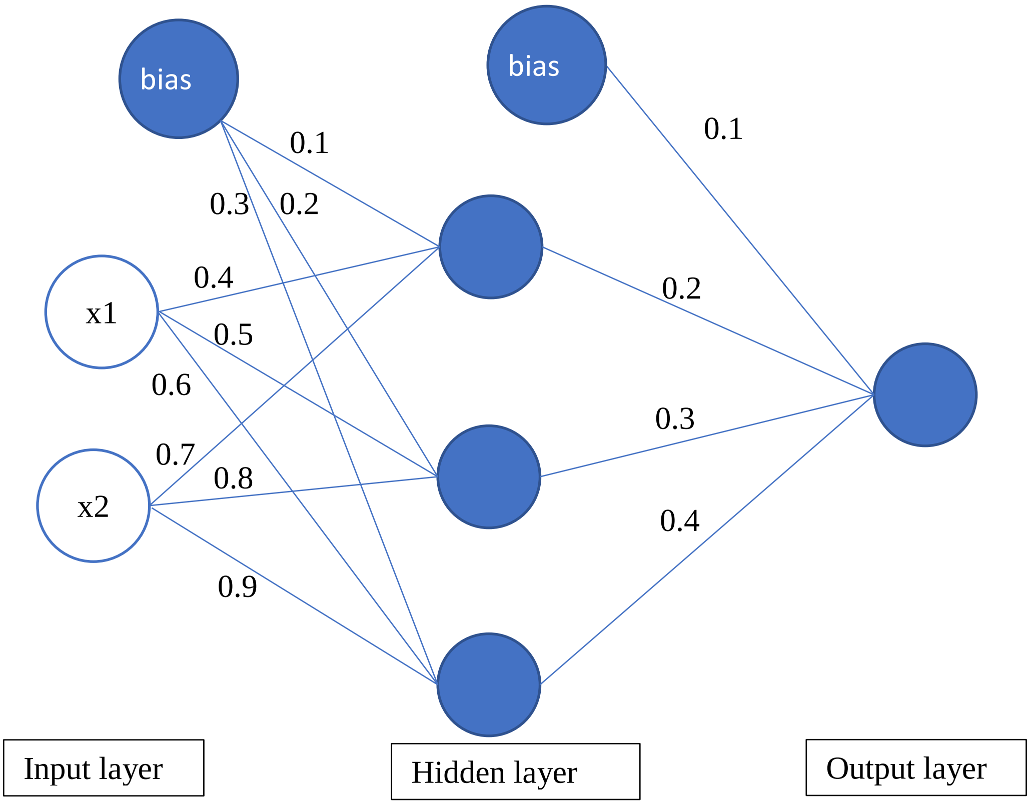
Accuracy: 6/30 = 0.2
Class 1:
P = 6/28
R = 6/8
Class 0:
P = R = 0

c)
In a situation where there is a large difference in size between the two classes, accuracy does not give the full picture. By putting all items in the majority class, the accuracy may be great, but the classifier is not very useful. In particular, this is a problem if we are mainly interested in the smallest class. Currently, it is estimated that 1% of the Norwegian population has been infected with covid-19. A test for antibodies that concludes for each person that she has not been infected, will be 99% accurate, but not very useful.

In some situations, it is more critical to misclassify items from one of the classes than from the other class. Consider spam classification. It is most important that I receive all my no-spams. Hence, if spam is considered the positive class of the classifier, we should go for a precision of 100%, on the cost of an imperfect recall.

5) Neural Networks (16 points)

We are using a feed forward neural network for solving a regression problem. The observation~~s~~ has m=2 features. There is one hidden-layer with n=3 nodes and a single output node. We assume that the logistic function is used as the activation function on the hidden layer. There is **no activation function on the output node** (since this is a regression problem.) At a point of time during training, the weights are as in the following figure.



a) Forwards step: Consider the observation **x** = (1, 2) What is the output value of the network for this observation? (5 p)

b) Backwards step: We assume we are training the network with stochastic gradient descent and will update the weights after each observation. The correct output for **x** in the training data is t = 10. Show how the weights are updated for this observation. Assume a learning rate of 0.1. Make additional assumptions when needed and explain them. (11 p)

Solution

See jupyter notebook solution at the end of the solutions.

The students are not required to write code – “all calculations can be done by hand”. However, they may use a calculator, spreadsheet, or – as we has done – code, as long as they show what they are doing. Spreadsheet, or other code, makes it easier to carry out the calculations as array operations.

For a)
Check the following

- Add bias at input (0.5p)
- Sum the inputs*V at hidden (1p)
- Applying logistic regression (1p)
- Adding bias to hidden output (0.5p)
- Sum the inputs*W at output (1p)
- Do *not* apply logistic function to sum at output node (1p)

For (b)
Comment: We have in class, mandatory assignment, and extra tutorial considered multi-layered NN for classification with several nodes in the output layer. This exercise represents a simplification in several respects, and the goal is for the students to show that they have a basic understanding such that they manage the simplifications. In particular

- Regression vs. classification (Several students asked whether t=10 was a typo). We have considered the relationship between regression and classification in detail when considering linear and logistic regression.
- A simplified error at the output node
- Only one output node

A rough guide for points:

- Correct error at output node (4p), while an error which is correct given wrong assumptions in the model (e.g. applying logistic function at output), max 2 p
- Correct error at hidden (given the error at output) (2p)
- Updating W (given the errors) (2p)
- Updating V (given the errors) (2p)
- Overall (1p)

6) Multi-class Classification (8 points)

a) What is meant by multi-class classification? (2p)

b) The goal is to classify pictures of fruit, where each picture contains one and only one kind of fruit. Say the classes are: apple, banana, grapes, orange, pear, plum. To prepare for machine learning, we have to represent these labels as numerical values. The first idea that comes to mind is to represent each class with an integer, say apple:0, banana:1, grapes:2, orange:3, pear:4, plum:5. Data sets are often stored and presented this way. Why isn’t it a good idea to use these numbers directly as targets in an ML system, say a feed forward neural network? (1p)

c) A better proposal is to use “one-hot encoding”, also called “one-out-of-n” encoding. Explain how that works and propose and encoding for the 6 classes. (2p)

d) One possible approach to multi-class classification is called “one vs. rest”. Explain how it works. (3p)

Solution

a) A multi-class classification task is one where there are more than two possible classes and each item belongs to exactly one class.

b) This will turn it into a regression problem which assumes that the class ‘grapes’ is more similar to ‘banana’ and ‘orange’ than ‘grapes’ is to ‘apple’ or ‘plum’. Since the order of the classes is random, or alphabetic, there is no reason to assume such a relationship.

c) We introduce 6 binary features. Each class is represented by a vector of length 6 where one feature equals 1 and the other five equals 0, e.g. say apple: [1,0,0,0,0,0] , banana: [0,1,0,0,0,0], grapes: [0,0,1,0,0,0] etc.

d) One vs rest is a strategy for multiclass classification. For each class, the training set is split into two: the items that belong to the class and all the others, and a binary classifier is trained on this set. We are using a learning algorithm that predicts probabilities for class membership, and the same learning algorithm for each split. To predict the value of a new item, one tries to classify it with each classifier and collect the probability scores, e.g. for an item x, the banana classifier may say that the probability of this being a banana is 0.1, the grapes classifier may say the probability of it being grapes is 0.2, and so on. The one vs. rest-classifier will choose the class for which the corresponding classifier ascribes the highest probability.

7) Data Scaling (8 points)

You want to classify patients with respect to the probability of developing coronary diseases. You have a training material with data from many patients where several risk factors are measured, including the blood concentration of

- Cholesterol
- Triglyceride
- Natrium

All of them are measured in mmol/liter.

You consider all your data and calculate the min, max, mean and standard variation for each of the three and for the values considered together, and get the following table (the numbers are realistic, but constructed for this exercise):

	Cholesterol	**Triglyceride**	**Natrium**	**Collected**
Min	2.5	0.2	120	0.2
Max	10.5	3.0	150	150
Mean	6	1.3	140	49.1
Std. dev	2	0.7	5	48

a) You have been advised to scale the data, and you choose to use min-max scaler. Explain the min-max scaler and show the result of scaling the following two observations. (2p)

A: (4.0, 2.0, 140)
B: (3.0, 0.1, 115)

b) You also want to see whether you get similar results by using the normal scaler (also called standard scaler). How is this different from the min-max scaler and what is the result of using this scaler to the two points? (2p)

c) Discuss the effect of scaling or not scaling your data with respect to the following three types of learners: *k* nearest neighbors (*k*NN), logistic regression, feed-forward neural network (MLP). (4p)

Solution

a)
We may scale to the interval [0, 1] or [-1, 1]. We show the solution with the former.
The formula is: x'= (x – min)/(max – min).
For A
x1' = (4 – 2.5)/(10.5 – 2.5) = 1.5/8 = 0.1875
x2' = (2 – 0.2)/(3 – 0.2) = 18/28 = 0.643
x3' = (140 – 120)/(150 – 120) = 2/3 = 0.667

For B
x1' = (3.0 – 2.5)/8 = 0.0625
x2' = (0.1 – 0.2)/2.8 = - 0.036
x3' = (115-120/30 = -1/6 = - 0. 167

Comment: Several students commented during the exam that there must be a mistake since the value for B/natrium is below min_natrium. The goal of supervised learning is to train a learner on some observed data, such that it can be applied to new and unseen data. There is no guarantee that new and unseen data is within the boundaries of observed data. This has been emphasized several times during the semester, and should be common knowledge.

b)
The formula is x'' = (x – mean)/s
For A
x1'' = (4.0 – 6)/2 = -1.0
x2'' = (2.0 – 1.3)/0.7 = 1.0
x3'' = (140 – 140)/5 = 0

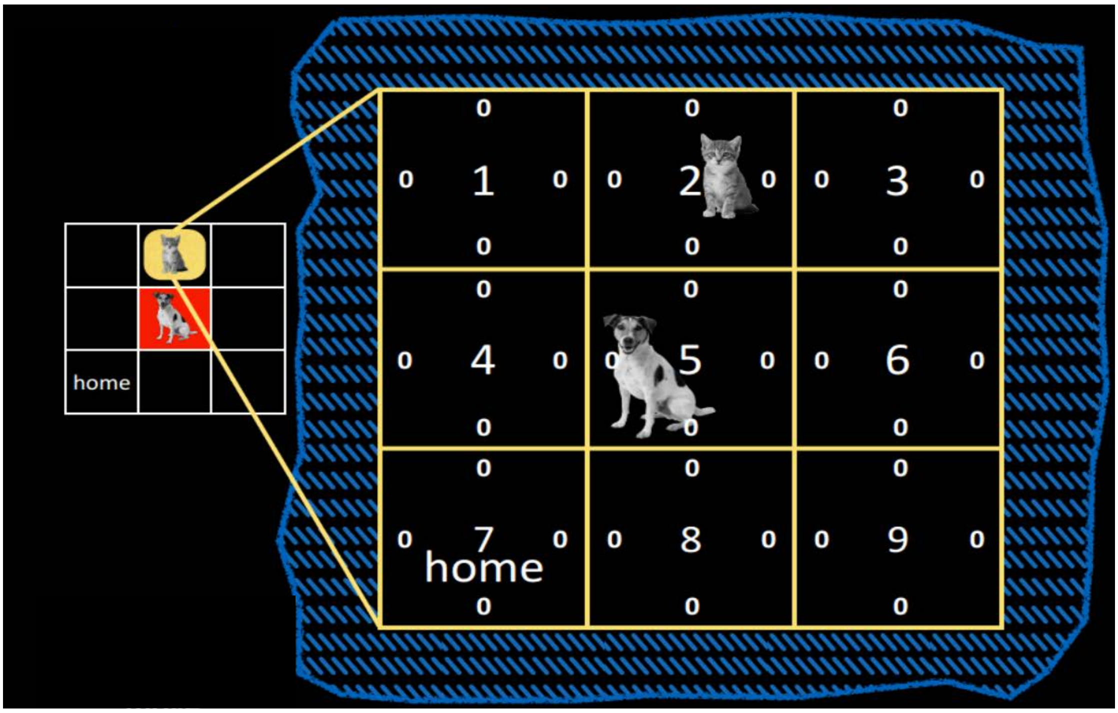
For B
x1'' = (3.0 – 6)/2 = -1.5
x2'' = (0.1 – 1.3)/0.7 = - 1.714
x3'' = (115 – 140)/5 = -5

c)
*k*NN Since *k*NN uses the distances between points, changing scales may alter the outcome, i.e. items may be classified differently with and without scaling. In the example, without scaling, natrium would be more important than the other features, because the distance between two points along this dimension will in general be larger than the distance along the other dimensions. (2p)

Logistic regression and MLP. Default values for the settings, e.g. learning rate is adapted to features in the interval [-1, 1] or in a an interval around 0 with standard deviation 1. If the features are much larger, e.g. 140, the gradient may not converge, (c.f. figure in solution to exercise 3 above). (1p)

One could accommodate this by choosing a smaller learning rate, but that does not fix the following problem. With a large difference between the different dimensions, the descent will primarily follow the steepest dimension, i.e. the one with the largest numerical features and – at least initially – not consider the other features. To take also the smaller features into considering, the learning may take a very long time. (1p)

Q-learning (23p)



Remember this example from the lecture slides on Reinforcement Learning (Lecture Week 12, Part 2)? The example describes a cat trying to make its way home without bumping into a dog, which we formulate as a reinforcement learning problem.

The squares are the possible locations the cat can visit (**states**), numbered 1 to 9. The numbers in the edges of the squares are q-values, Q(s,a) describing the value of performing each possible **action** (move left, right, down, up) in that state. They are initialized to 0, and by using q-learning we want to update them to more accurately reflect the value of taking each action in each state.

If the cat moves outside the area defined by the 9 squares (e.g. by choosing the "up" action in State 2), it "falls off the table", and gets a negative reward, before being moved back to the state he came from.

The reward structure of the problem is as follows – **note that it is different from the reward structure in the original example from class**:

- Each movement by the cat gives a reward of -0.5 to encourage the cat to be efficient. Note that this reward is not applied when any of the other reward conditions below are present.
- Unlike in the lecture, this cat likes dogs, so it gets a reward of +5 for moving to the state of the dog (state 5).
- The cat gets a reward of -1 for "falling off the table".
- The cat gets a reward of +5 for moving to the "home" state (state 7).

We will use a discount factor of 0.9 and a learning rate of 0.2 (note: This is different from the example in the lecture).

a) Fill in the q-values of all states after the cat has performed 8 actions and updated the values with q-learning. The cat starts in state 2, and performs the following actions, receiving rewards and moving to new states as defined above. Action sequence: [Down, Right, Up, Right, Left, Down, Right, Down]. A reminder of the q-value calculation:

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\mu}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

Show the intermediate calculations. That is, if you update a given q-value twice, show the value after each update. (5p)

b) The second time the cat performs the "Down" action in state 2, the Q-value changes – even though the cat ends up in the same state as the previous time, and it gets the same reward. Why does it change? What would happen if the cat performed the "Down" action in state 2 very many times? Would the Q-value converge towards a specific value – and if so, which value? (you don't need to prove this mathematically, but explain the intuitions with your own words)
(3p)

c) With the q-values filled in after completing task a, what is the likelihood of choosing the action "down" in State 2, given you are using the following policies: 1) A greedy policy? 2) An epsilon-greedy policy with epsilon equal to 0.9? 3) A soft-max policy with a temperature of 1? Show your calculations and/or justify your answers.

Rank the policies with regards to how much they explore, and justify your ranking.
(4p)

d) What is the role of the discount factor? Why do we need to have one? What would happen to your calculated q-values if we reduced the discount factor to 0.1? What assumption do we make when we use a discount factor of 1?
(4p)

e) Assume we reset all q-values to 0. Perform the same actions, but update q-values according to the SARSA algorithm. You can assume that the actions in the sequence were actually the actions decided by the cat's policy. Since SARSA requires to know the consecutive action for updating q-values, you should perform only 7 updates of q-values this time, that is, do not try to update the q-value of the final "Down" action. Which q-values, will be different from the result in a?

For the values that are different, does off-policy or on-policy learning give the highest q-value? Referring to the assumptions made by q-learning on future rewards: why does on-policy learning and off-policy learning give different results here?
(5p)

f) We have made a little mistake in our reward function. Can you identify a form of "reward hacking" the cat can do, where it displays unwanted behaviour but still receives a high reward? (it is NOT visiting the dog – this cat really does like dogs)
(2p)

Solution

a) Q(2, Down) = 0 + 0.2*(5 + 0.9*0 – 0) = 1
Q(5,Right) = 0 + 0.2*(-0.5 + 0.9*0 – 0) = -0.1
Q(6,Up) = 0 + 0.2*(-0.5 + 0.9*0 – 0) = -0.1
Q(3,Right) = 0 + 0.2*(-1 + 0.9*0 – 0) = -0.2
Q(3, Left) = 0 + 0.2*(-0.5 + 0.9*1 – 0) = 0.08
Q(2,Down) = 1 + 0.2*(5 + 0.9*0 – 1) = 1.8
Q(5,Right) = -0.1 + 0.2*(-0.5 + 0.9*0 – (-0.1)) = -0.18
Q(6, Down) = 0 + 0.2*(-0.5 + 0.9*0 – 0) = -0.1

(5p)

b) Since the learning rate is less than 1, just a little information about the reward and next optimal q-value is propagated to the Q-value on every iteration. Therefore, observing the same reward twice moves the Q-value a little bit more towards the true value of the expected future reward of the state/action pair. (2p)
The Q-value will move slowly towards 5 (the actual reward) if repeating this action many times. (1p)

c) 1) A greedy policy would choose action “Down” since it has the highest q-value. (1p)
2) The softmax function is:

$$P\left(Q_{s,t}(a)\right) = \frac{e^{(Q_{s,t}(a)/\tau)}}{\sum_b e^{(Q_{s,t}(b)/\tau)}}$$

With all q-values but Q(down) being 0, we get:
$e^{1.8}$ = 6.05. e^{0} = 1.
6.05/(1+1+1+6.05) = 0.67 (1p)

3) Epsilon-greedy has a 0.1 probability of choosing the optimal action here, and a 0.9 probability of exploring randomly. When exploring randomly, there is a 0.25 chance of selecting “Down”. The full probability of selecting “Down” is thus:
0.1 + 0.9*0.25 = 0.325
(Note that this epsilon-value is far higher than what should be used in practice, as it makes the agent act extremely randomly.) (1p)

Exploration amount in increasing order: Greedy – softmax - epsilon-greedy.
Greedy explores the least, since it has no randomness – always chooses the optimal action. Epsilon-greedy explores the most, since it has the lowest probability of choosing the optimal action (due to the very high epsilon-value), giving higher probabilities for exploring the sub-optimal ones.
(1p)

d) The discount factor reduces the impact of future expected rewards when calculating the q-value of the current state. (1p)
It is important to have one when the future is a bit uncertain, since when we have some uncertainty, immediate rewards are more valuable. (1p)
If we reduced the impact factor, states where we have high expected future rewards would be valued a bit less. In the example above, the only situation where we have a higher than 0 expected future reward is when going left in state 3. With a discount

factor of 0.1, we would there end up with a negative q-value despite there being a nice future reward waiting in state 2. (1p)

With a discount factor of 1, we are assuming that the future is completely certain, or at least that we don't care about any uncertainty about the future in our calculations. (1p)

e) SARSA Q-update function:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \mu[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Notice that *unlike q-learning, SARSA updates do not assume the optimal future reward from the next state, but rather that we get the reward associated with the action the agent actually makes in that state (governed by its policy).* (1.5p)

For most calculations in this example, the two have the same result, since the action the agent takes and the optimal action in a state both have a Q-value of 0 before the agent has explored the environment. *The only q-values that will be different are the ones resulting from visiting states we have been to before, where the optimal q-value is not the same as the q-value from the chosen action*. Only the following q-value will be different:
Q(2,Down) = 1 + 0.2*(5 + 0.9*(-**0.1**) – 1) = 1.78

The exam question read "values", but should have said value(s), so if students have by mistake added one additional q-value that becomes different, they can still get a full score since the question text may have tricked them.
(2p)

The **-0.1 in bold** are the q-values of the actual chosen next action, rather than the optimal action which had a q-value of 0. *Since the actual chosen action was not optimal, this on-policy q-estimate has a lower value than the off-policy estimate from q-learning.*
(1.5p)

f) When the cat gets to the home state, it can stay there and *keep hopping off the table, accumulating a high reward even though this is a silly behaviour*. Full score can be given if students find other "bad" behaviours that by mistake turn out to give a high reward.

8) Particle Swarm Optimization (6p)

a) Explain how particles' position and velocity are updated in each iteration with reference to the formulas below. (2p)

$$x_{i,d}(it+1) = x_{i,d}(it) + v_{i,d}(it+1) \qquad (1)$$

$$\begin{aligned} v_{i,d}(it+1) &= v_{i,d}(it) \\ &+ C_1 * Rnd(0,1) * [pb_{i,d}(it) - x_{i,d}(it)] \\ &+ C_2 * Rnd(0,1) * [gb_d(it) - x_{i,d}(it)] \end{aligned} \qquad (2)$$

b) What happens if the information of the global best solution is not used in PSO? (2p)

c) PSO and evolutionary algorithms (EAs) are population-based search methods that have quite a lot of things in common. 1) In an EA, what concept corresponds to the particles in PSO? 2) In an EA, what concept corresponds to the positional parameters in PSO? 3) And finally, in an EA, what concept corresponds to/is related to the position update rule in PSO? Briefly justify your answers and any assumptions. (2p)

Solution

a) Students should be able to describe the process of updating the position based on the velocity (0.5p). How the velocity is updated is unique to PSO. They should mention that the best solution found (pbest) in a particle and the global best solution found (gbest) will be used to udpate the velocity (1p). In addition, the two acceleration constants determine how much pbest and gbest update the velocity [this leads to an exploration vs exploitation trade-off (0.5p).

b) Students should realize that the global best solution is the best position of the particle that has been found. If this information is not used in PSO, the particles will do a hill climbing local search *in the direction of their velocity vectors* (2p).

c) When making an analogy of PSO with evolutionary algorithms, a particle is similar to an individual, the positional parameters are similar to genes/genotype/evolvable parameters/mutable parameters/ chromosome, etc. and the weight update rule is related to / similar to the variation operators in EAs (especially mutation) although EA variation operators have a lot more randomness. (0.67p for each sub-question)

Exercise_5_Neural_Networks

June 6, 2020

```
[1]: import numpy as np
     import matplotlib.pyplot as plt
     import sklearn
```

0.1 Initialize

```
[2]: def logistic(x):
         return 1 / (1 + np.exp(-x))
```

```
[3]: V = np.arange(1,10).reshape([3,3]) * 0.1
```

```
[4]: V
```

```
[4]: array([[0.1, 0.2, 0.3],
            [0.4, 0.5, 0.6],
            [0.7, 0.8, 0.9]])
```

```
[5]: W = np.arange(1, 5) * 0.1
     W
```

```
[5]: array([0.1, 0.2, 0.3, 0.4])
```

```
[6]: eta = 0.1
```

```
[7]: x = np.array([-1, 1, 2]) # x[0] er bias
```

```
[8]: target = 10
```

Comment Since we are only considering one datapoint, I have simplified and represented it as an array. For a more general solution also covering batch-processing, x should be reshaped into an matrix of shape 1*3.

0.2 Forward

Taking the sum of the input on each node in hidden.

```
[9]: z = x @ V
     z
```

```
[9]: array([1.7, 1.9, 2.1])
```

Applying the activation function.

```
[10]: a = logistic(z)
      a
```

```
[10]: array([0.84553473, 0.86989153, 0.89090318])
```

Adding bias

```
[11]: a_biased = np.concatenate([np.array([-1]), a])
      a_biased
```

```
[11]: array([-1.        ,  0.84553473,  0.86989153,  0.89090318])
```

Calculating output

```
[12]: out = a_biased @ W
      out
```

```
[12]: 0.6864356761961485
```

0.3 Backpropagation

To calculate the loss at the output node, we are using 1/2 Squared Error, like Marsland. Mean squared error (with or without multiplying with 1/2) works equally well, but may yield different numbers for one backprop step.

The loss at output is simple since the "activation" is identity.

```
[13]: out_error = (target - out)
      out_error
```

```
[13]: 9.31356432380385
```

Projecting the error back to the hidden layer.

This is also unusually simple since there is only one connection going out of each hidden node.

```
[14]: delta_back = W * out_error
      delta_back
```

```
[14]: array([0.93135643, 1.86271286, 2.7940693 , 3.72542573])
```

```
[15]: hidden_deltas = delta_back[1:] * a * (1-a)
      hidden_deltas
```

```
[15]: array([0.24328101, 0.31623349, 0.36209165])
```

Updating W

```
[16]: W += eta * out_error * a_biased
      W
```

```
[16]: array([-0.83135643,  0.98749421,  1.11017907,  1.22974841])
```

Updating V

So far, it has worked well to use arrays for x and (x @ V). However, to easily update the weights, we have to turn them into matrices and transpose x.

```
[17]: x.reshape([-1, 1]) @ hidden_deltas.reshape([1, -1])
```

```
[17]: array([[-0.24328101, -0.31623349, -0.36209165],
             [ 0.24328101,  0.31623349,  0.36209165],
             [ 0.48656201,  0.63246697,  0.72418331]])
```

```
[18]: V  += eta * x.reshape([-1, 1]) @ hidden_deltas.reshape([1, -1])
      V
```

```
[18]: array([[0.0756719 , 0.16837665, 0.26379083],
             [0.4243281 , 0.53162335, 0.63620917],
             [0.7486562 , 0.8632467 , 0.97241833]])
```

```
[ ]:
```

Exercise_5_Neural_Networks_bias_1

June 9, 2020

```
[1]: import numpy as np
     import matplotlib.pyplot as plt
     import sklearn
```

0.1 Initialize

```
[2]: def logistic(x):
         return 1 / (1 + np.exp(-x))
```

```
[3]: V = np.arange(1,10).reshape([3,3]) * 0.1
```

```
[4]: V
```

```
[4]: array([[0.1, 0.2, 0.3],
            [0.4, 0.5, 0.6],
            [0.7, 0.8, 0.9]])
```

```
[5]: W = np.arange(1, 5) * 0.1
     W
```

```
[5]: array([0.1, 0.2, 0.3, 0.4])
```

```
[6]: eta = 0.1
```

```
[7]: x = np.array([1, 1, 2]) # x[0] is bias
```

```
[8]: target = 10
```

Comment Since we are only considering one datapoint, I have simplified and represented it as an array. For a more general solution also covering batch-processing, x should be reshaped into an matrix of shape 1*3.

0.2 Forward

Taking the sum of the input on each node in hidden.

```
[9]: z = x @ V
     z
```

```
[9]: array([1.9, 2.3, 2.7])
```

Applying the activation function.

```
[10]: a = logistic(z)
      a
```

```
[10]: array([0.86989153, 0.90887704, 0.93702664])
```

Adding bias

```
[11]: a_biased = np.concatenate([np.array([1]), a])
      a_biased
```

```
[11]: array([1.        , 0.86989153, 0.90887704, 0.93702664])
```

Calculating output

```
[12]: out = a_biased @ W
      out
```

```
[12]: 0.9214520744001451
```

0.3 Backpropagation

To calculate the loss at the output node, we are using 1/2 Squared Error, like Marsland. Mean squared error (with or without multiplying with 1/2) works equally well, but may yield different numbers for one backprop step.

The loss at output is simple since the "activation" is identity.

```
[13]: out_error = (target - out)
      out_error
```

```
[13]: 9.078547925599855
```

Projecting the error back to the hidden layer.

This is also unusually simple since there is only one connection going out of each hidden node.

```
[14]: delta_back = W * out_error
      delta_back
```

```
[14]: array([0.90785479, 1.81570959, 2.72356438, 3.63141917])
```

```
[15]: hidden_deltas = delta_back[1:] * a * (1-a)
      hidden_deltas
```

```
[15]: array([0.20550248, 0.22556442, 0.21428174])
```

Updating W

```
[16]: W += eta * out_error * a_biased
      W
```

```
[16]: array([1.00785479, 0.98973519, 1.12512838, 1.25068413])
```

Updating V

So far, it has worked well to use arrays for x and (x @ V). However, to easily update the weights, we have to turn them into matrices and transpose x.

```
[17]: x.reshape([-1, 1]) @ hidden_deltas.reshape([1, -1])
```

```
[17]: array([[0.20550248, 0.22556442, 0.21428174],
             [0.20550248, 0.22556442, 0.21428174],
             [0.41100496, 0.45112884, 0.42856348]])
```

```
[18]: V  += eta * x.reshape([-1, 1]) @ hidden_deltas.reshape([1, -1])
      V
```

```
[18]: array([[0.12055025, 0.22255644, 0.32142817],
             [0.42055025, 0.52255644, 0.62142817],
             [0.7411005 , 0.84511288, 0.94285635]])
```

```
[ ]:
```