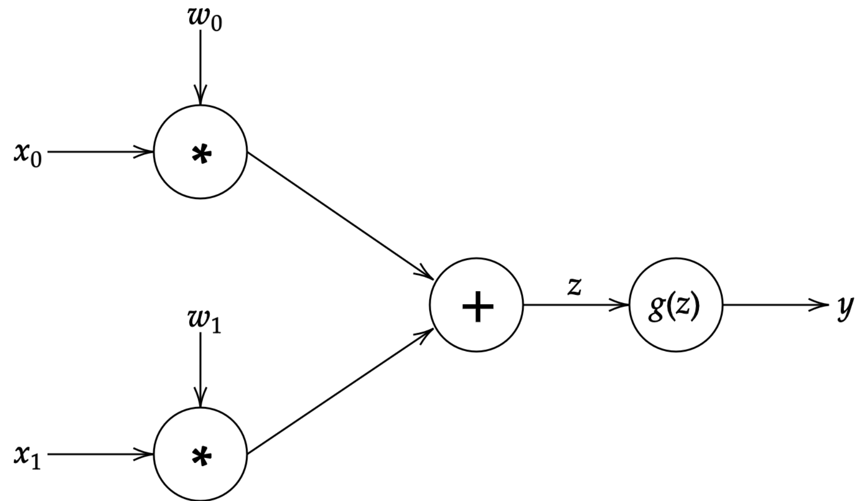
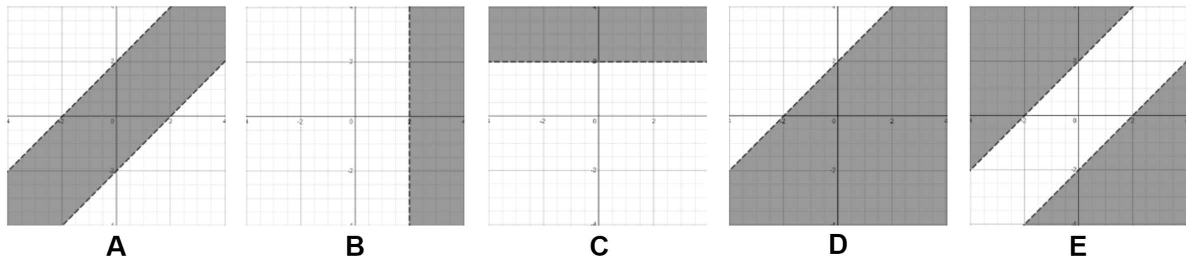
Q7. [8 pts] So Many Derivatives

Consider the neural network configuration below.



(a) [2 pts] Which of the following decision boundaries can be learned by the neural network? Assume $w_0, w_1, x_0, x_1, T_a \in \mathbb{R}^n$ and $z = w_0 x_0 + w_1 x_1$. Let $g(z)$ be the binary step activation function with T_a as the decision threshold, which is defined as follows:

$$g(z) = \begin{cases} 1 \text{ if } z \geq T_a \\ 0 \text{ if } z < T_a \end{cases}$$



- [ ] Graph A
- [x] Graph B
- [x] Graph C
- [x] Graph D
- [ ] Graph E
- ○ None of the above

(b) Now let $g(z)$ be the sigmoid activation function and y be a real number value between 0 and 1 (we will ignore the threshold T_a for this part). Recall that the derivative of the sigmoid function is $\frac{\partial}{\partial z} g(z) = g(z) \cdot (1 - g(z))$. You can represent your answers in terms of x_0, x_1, w_0, w_1, z, or y.

(i) [2 pts] Calculate the following partial derivatives for backpropagation.

(1) $\frac{\partial y}{\partial z} = $ $y(1-y)$

(2) $\frac{\partial z}{\partial w_0} = $ x_0

(ii) [2 pts] Suppose we are running gradient **descent** on the neural network above. We are trying to minimize the upstream loss L using learning rate α. Given the upstream gradient $\frac{\partial L}{\partial y}$ and the two partial derivatives that you computed in the previous part ($\frac{\partial y}{\partial z}$ and $\frac{\partial z}{\partial w_0}$), determine the gradient descent update rule for w_0.

$w_0 \leftarrow$ $w_0 - \alpha \cdot \frac{\partial L}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_0}$

(c) [2 pts] The Binary Perceptron is defined as the following:

$$y = \text{classify}(x) = \begin{cases} +1 & \text{if } w \cdot f(x) + b \geq 0 \\ -1 & \text{if } w \cdot f(x) + b < 0 \end{cases}$$

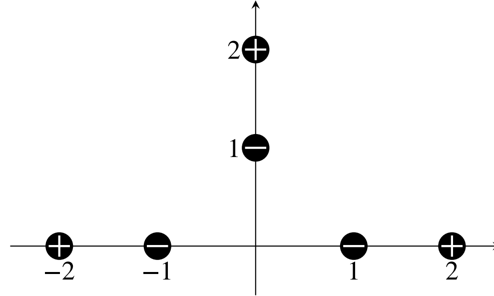
where w is a vector of real-valued weights, $w \cdot f(x)$ is the dot product $\sum_{i=1}^{m} w_i f_i(x)$ where m is the number of features, $f_i(x)$ is the ith feature of x, and b is the bias.

Which of the following are true about the binary perceptron as defined above?

- [ ] It is possible that the perceptron learns a decision boundary that is nonlinear in terms of the features $f(x)$.
- [x] It is possible that the perceptron learns a decision boundary that is nonlinear in terms of the data x.
- [x] The perceptron algorithm is guaranteed to converge if the data is linearly separable.
- [ ] The perceptron algorithm is trained using gradient descent.
- ○ None of the above

Q9. [8 pts] Higher-Dimensional Perceptrons

Consider a dataset with 6 points on a 2D coordinate grid Each point belongs to one of two classes. Points $[-1, 0], [1, 0], [0, 1]$ belong to the negative class. Points $[-2, 0], [2, 0], [0, 2]$ belong to the positive class.



(a) [1 pt] Suppose we run the perceptron algorithm with the initial weight vector set to $[0, 5]$.

What is the updated weight vector after processing the data point $[0, 1]$?

$[0, 4]$

We have $f = [0, 1]$ and $w = [0, 5]$. First we classify the point by computing the dot product: $f \cdot w = 5$. This classifies f in the positive class, but the true class is negative.

Our classification is wrong, so we need to adjust the weights by subtracting the feature vector: $w - f = [0, 5] - [0, 1] = [0, 4]$.

(b) [1 pt] How many iterations of the perceptron algorithm will run before the algorithm converges? Processing one data point counts as one iteration. If the algorithm never converges, write ∞.

∞

The data points are not linearly separable, so the perceptron algorithm will never terminate.

In the next few subparts, we'll consider *transforming* the data points by applying some modification to each of the data points. Then, we pass these modified data points into the perceptron algorithm.

For example, consider the transformation $[x, y] \to [x, y, x^2, 1]$. In this transformation, we add two extra dimensions: one whose value is always the square of the first coordinate, and one whose value is always the constant 1. For example, the point at $[2, 0]$ is transformed into a point at $[2, 0, 4, 1]$ in 4-dimensional space.

(c) [2 pts] Which of the following data transformations will cause the perceptron algorithm to converge, when run on the transformed data? Select all that apply.

- ☐ $[x, y] \to [y, x]$
- ☐ $[x, y] \to [x, y, 1]$
- ■ $[x, y] \to [x, y, x^2, 1]$
- ☐ $[x, y] \to [x, y, x^2 + y^2]$
- ● None of the above.

(A): False. Pictorially, this transformation reflects all the data points across the $x = y$ line. If you graph the resulting points, they're still not linearly separable, so the perceptron algorithm still never terminates.

(B): Pictorially, this transformation plots the data points along a flat plane in the 3D grid. If you graph the resulting points, they're still not linearly separable, so the perceptron algorithm still never terminates.

If you don't want to picture points in higher dimensions, another solution is to note that the resulting decision boundary here is $w_1 x + w_2 y + w_3 > 0$. In other words, you added a y-intercept term w_3 to the decision boundary line on the 2D coordinate plane. Adding a y-intercept so that the decision boundary doesn't have to cross the origin still doesn't help us linearly separate the data, though.

(C): True. Write the decision boundary equation $w_1x + w_2y + w_3x^2 + w_4 > 0$, and note that this is some form of parabola (quadratic equation) in the 2D coordinate plane, since we have terms with y, x^2, x, plus some constant.

Intuitively, you could sketch a parabola that crosses coordinate points $[-1.5, 0]$, $[0, 1.5]$, and $[1.5, 0]$ on the coordinate grid, which would separate the points.

If you wanted to find the exact equation of this line (which was not necessary for this question), you could start with $y = x^2$. Then note that the parabola has to point down, so it should be something like $y = -x^2$. Then note that we should shift this parabola upwards so that the negative points are "below" the parabola, to get something like $y = -x^2 + 1.5$. Rearranging gives the decision boundary $y + x^2 - 1.5 > 0$.

You can confirm that this equation works by plugging in all six points and noting that the negative points all have $y + x^2 - 1.5 < 0$, and all the positive points have $y + x^2 - 1.5 > 0$.

(D): False. The new feature, $x^2 + y^2$, is equal to 1 for all the negative points and 4 for all the positive points. However, the perceptron classifies points based on the sign of the output, so we'd have somehow use the remaining features to map all the 1s to negative numbers, and all the 4s to positive numbers. We don't have a constant (like in the previous options) to help us with this, and trying to add/subtract multiples of x or y proves to not be useful either (playing around with a few possibilities should be enough to convince you that x and y can't help here).

A geometric solution (which is not necessary to solve this problem, but useful if you like thinking geometrically): note that the $x^2 + y^2$ term looks like the equation of a circle, so adding this term introduces circles into our decision boundaries. We can draw circles and classify points inside the circle as one class, and points outside the circle as a different class. However, we lack a constant term, so our decision boundary is always going to be in the form $w_1x + w_2y + w_3(x^2 + y^2) > 0$. We can see that $[0, 0]$ is always going to fall on the decision boundary, so the lack of constant term restricts our decision boundaries to only the circles that pass through the origin. Visually, we can see that a circle passing through the origin will not separate the points.

(d) [2 pts] Suppose we transform $[x, y]$ to $[x, y, x^2 + y^2, 1]$, and pass the transformed data points into the perceptron.

Write one possible weight vector that the perceptron algorithm may converge to.

[0,0,1,-2]

The new $x^2 + y^2$ coordinate looks very useful for separating the data. Note that for the negative points, the new coordinate is always 1, and for the positive points, the new coordinate is always 4.

However, 1 and 4 are both positive, and the perceptron classifies based on the sign of the output. To fix this, we need to use the constant bias term to add any negative bias $-4 < c < -1$ from every classification so that 1 maps to some negative number and 4 maps to some positive number. For example, $c = -2$ would map positive points to $1 - 2 = -1$ and negative points to $4 - 2 = 2$.

If you used a different weight k for the $x^2 + y^2$ feature, you would get perceptron activation of k for all the negative points and $4k$ for all the positive points. Then, your constant factor would need to be in the range $-4k < c < -k$ so that k gets mapped to a negative number, and $4k$ gets mapped to a positive number.

A nice geometric solution (which is not necessary to solve this problem): the $x^2 + y^2$ feature, along with the constant bias, lets us draw circular decision boundaries. The restriction to circles passing through the origin, from the previous subpart, no longer applies here because we've introduced a constant bias. Now, we can draw a circle that separates the points. Any circle with center at the origin, and radius between 1 and 2, will perfectly separate the points (all negative points inside, all positive points outside). If we set the radius to be 1.5, we'd get the circle equation $x^2 + y^2 = 1.5^2 = 2.25$. Rearranging terms a bit, we get the decision boundary $x^2 + y^2 - 2.25 > 0$. This corresponds to the weight vector $[0, 0, 1, -2.25]$, which is also a correct solution.

(e) [2 pts] Construct another transformation (not equal to the ones above) that will allow the perceptron algorithm to converge.

Hint: The transformation $[x, y] \to [x, y, x^2 + y^2, 1]$ allows the perceptron algorithm to converge.

Fill in the blank: $[x, y] \to [x, y, ___, 1]$.

$x^4 + y^4$

One simple class of transformations that works here is any that combines the magnitudes of the two coordinates. (Pictorially, this corresponds to the fact that the negative points are closer to the origin, and the positive points are further away from the origin.) Some sample answers include: $x^4 + y^4$, or $x^6 + y^6$, or $|x| + |y|$, etc.

Another simple class of transformations is to add a constant factor to the x^2+y^2 feature that helped from earlier: $2(x^2+y^2)$, or $3(x^2+y^2)$, or $4(x^2+y^2)$, etc. These transformations still work because you could always adjust the third weight value from the $[x, y, x^2+y^2, 1]$ perceptron to cancel out the new coefficient, which would give you back the original decision boundary that worked on the $[x, y, x^2+y^2, 1]$ perceptron. For example, if your transformation is $x^2+y^2 \rightarrow c(x^2+y^2)$ and the original weight vector had w_3, you could adjust the weight vector to w_3/c and end up with the same decision boundary.

Another simple class of transformation is to add a constant value to the x^2+y^2 feature from earlier: x^2+y^2+1, x^2+y^2+2, etc. If your transformation is $x^2+y^2 \rightarrow x^2+y^2+c$, then you've added a constant value cw_3 to every activation value. If you adjust the constant bias weight value from w_4 to $w_4 - cw_3$, then you cancel out the new addition and end up with the same original decision boundary.

Other solutions probably exist here, but these were the simplest three that we could think of.

Exam continues on next page.

Q10. [9 pts] Q-Networks

Consider running Q-learning on the following Pacman problem: the maze is an x-by-x square, and each position can contain a food pellet or no food pellet. There are no ghosts or walls. Pacman's only actions are $\{\text{up}, \text{down}, \text{left}, \text{right}\}$.

(a) [2 pts] How many Q-values do we need to learn for this problem?

Your answer should be an expression, possibly in terms of x.

$4x^2 \cdot 2^{x^2}$

The table represents $Q(s, a)$, so we need $|S| \times |A|$ entries. $|A| = 4$. Pacman has x^2 possible locations. Every location has two potential statuses (food or empty). Hence the table size is $4x^2 \cdot 2^{x^2}$.

Recall that in Q-learning, we maintain a table of $Q(s, a)$ values, where there's one Q-value for every state-action pair.

There are 4 actions available from any given state. (Note: We got a few clarification questions during the exam about actions that would cause Pacman to leave the maze. Here, we either assumed that these actions were legal but left Pacman in the same position, or, if these actions were illegal, that 4 is a reasonable upper-bound on the number of available actions from any given state.)

The state space should include Pacman's position, and a list of Booleans indicating whether each position has a food pellet or not. There are x^2 possible locations for Pacman. There are x^2 Booleans we have to keep track of, so there are 2^{x^2} possible configurations of food pellets. In total, the problem has $x^2 \cdot 2^{x^2}$ possible states.

For each of the $x^2 \cdot 2^{x^2}$ states, there are 4 possible actions. Therefore, there are $4x^2 \cdot 2^{x^2}$ state-action pairs.

To learn every Q-value, we could run standard Q-learning, but we decide to try a different approach:

Suppose somebody tells us N exact Q-values for N different state-action pairs: the exact Q-value for the state-action pair (s_i, a_i) is q_i, for $1 \leq i \leq N$. (N is less than the total number of state-action pairs.)

We decide to use these exact Q-values to train a neural network, so that we can estimate other Q-values we don't know. To train this neural network, we need to apply gradient descent to minimize the following loss function:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} (f(\theta, s_i, a_i) - q_i)^2$$

θ represents the weights of the neural network. $f(\theta, s_i, a_i)$ represents running the neural network with weights θ on state-action pair (s_i, a_i).

(b) [1 pt] What is the gradient $\frac{\partial L}{\partial \theta}$?

Your answer should be an expression, possibly in terms of N, $\frac{\partial f}{\partial \theta}$, and q_i.

$\frac{\partial L}{\partial \theta} =$ $\frac{2}{N} \sum_{i=1}^{N} \left[\frac{\partial f}{\partial \theta} (f(\theta, s_i, a_i) - q_i) \right]$

Clarification during exam: Your expression could also use $f(\theta, s_i, a_i)$ and q_i.

Take the partial derivative using the chain rule.

The 2 comes from applying the power rule on the square of each term in the summation, and then factoring it out. The $1/N$ constant coefficient comes from the coefficient in the original expression. The summation stays because the derivative of a sum is equal to the sum of the derivatives.

Inside the summation, we're taking the derivative with respect to θ, so by the chain rule, we need to have a factor of $\frac{\partial f}{\partial \theta}$.

(c) [1 pt] After running t iterations of gradient descent, our current weights are θ_t. The learning rate is α.

What are the weights on the next iteration, θ_{t+1}?

Your answer should be an expression, possibly in terms of θ_t, α, and $\frac{\partial L}{\partial \theta}$.

$\theta_{t+1} =$ $\theta_t - \alpha \frac{\partial L}{\partial \theta}$

This is the expression for gradient descent from lecture. We take the current weights θ_t, and move them in the direction of the gradient $\frac{\partial L}{\partial \theta}$. We apply a learning rate of α, and we use subtraction because gradient descent involves moving in the opposite direction of the gradient.

(d) [2 pts] Eventually, gradient descent converges to the weights θ^*.

We use the neural network with weights θ^* to compute Q-values, and extract a policy out of these Q-values:

$$\pi(s) = \arg\max_a f(\theta^*, s, a)$$

Is π the optimal policy for this problem?

○ Yes ○ No ● Not enough information

When we train a neural network on some training data (here, some subset of all the Q-values) and then use the network to classify some unseen test data (here, the unseen Q-values), there is no guarantee that we are going to perfectly predict the test data values. In other words, test accuracy in a neural network is not guaranteed to be perfect.

It's possible (but unlikely) that we get lucky and our neural network perfectly predicts all unseen Q-values. It's also possible that our neural network makes some errors when predicting unseen Q-values. We don't have enough information to know what the test accuracy of the neural network is.

Instead of the Pacman problem, consider a different problem where the action space is continuous. In other words, there are infinitely many actions available from a given state.

(e) [3 pts] Can we still use the strategy from the previous subparts (without any modifications) to obtain a policy π?

○ Yes ● No

Briefly explain why or why not.

When we move to a continuous action space, the main part of our strategy that breaks is the policy extraction step $\pi(s) = \arg\max_a f(\theta^*, s, a)$.

When we had a finite number of actions, this argmax involved trying each value of a and picking the one with the highest $f(\theta^*, s, a)$ value. However, when we have an infinite number of actions available, this argmax becomes more difficult (or even impossible).

The neural network step should still mostly work; we can still pass in some existing training data and learn weights. Then, we can still use the neural network model to predict Q-values of state-action pairs that we've never seen before, even if there are infinitely many actions.