COMS 4771 HW2

Due: Mon March 18, 2019 at 11:59pm

You are allowed to write up solutions in groups of (at max) three students. These group members don't necessarily have to be the same from previous homeworks. Only one submission per group is required by the due date on Gradescope. Name and UNI of all group members must be clearly specified on the homework. No late homeworks are allowed. To receive credit, a typesetted copy of the homework pdf must be uploaded to Gradescope by the due date. You must show your work to receive full credit. Discussing possible solutions for homework questions is encouraged on piazza and with peers outside your group, but every group must write their own individual solutions. You should cite all resources (including online material, books, articles, help taken from specific individuals, etc.) you used to complete your work.

1 **[Constrained optimization]** Show that the distance from the hyperplane $g(x) = w \cdot x + w_0 = 0$ to a point x_a is $|g(x_a)|/\|w\|$ by minimizing the squared distance $\|x - x_a\|^2$ subject to the constraint $g(x) = 0$.

2 **[Data dependent perceptron mistake bound]** In class we have seen and proved the perceptron mistake bound which states that the number of mistakes made by the perceptron algorithm is bounded by $\left(\frac{R}{\gamma}\right)^2$.

(i) Prove that this is tight. That is, give a dataset and an order of updates such that the perceptron algorithm makes exactly $\left(\frac{R}{\gamma}\right)^2$ mistakes.

Interestingly, although you have hence proved that the perceptron mistake bound is tight, this does not mean that it cannot be improved upon. The claimed "tightness" of the bound simply means that there exists a "bad" case which achieves this worst case bound. If we make some extra assumptions, these bad cases might be ruled out and the worst case bound could significantly improve. In ML, it is common to look at how extra assumptions can help improve such bounds [1]. This is what we will do in this problem.

As in class let $S = \{(x_i, y_i)\}_{i=1}^n$ be our (linearly separable) dataset where $x_i \in \mathbb{R}^D$ and $y_i \in \{-1, 1\}$. Also let w^* be the unit vector defining the optimal linear boundary with the optimal margin γ (i.e. $\forall i, y_i(w^* \cdot x_i) \geq \gamma$). Finally, let $R = \max_{x_i \in S} \|x_i\|$. Note that the standard bound tells us that the perceptron algorithm will make at most $\left(\frac{R}{\gamma}\right)^2$ mistakes.

Now assume that we are given the extra information that $\max_{x_i \in S} \|(I - P)x_i\| \leq \epsilon < R$ where $P = w^* {w^*}^\mathsf{T}$ and thus $(I - P)$ is the projector onto the orthogonal complement space

[1] Indeed there is a vast field that comprises of trying to get "data-dependent bounds", i.e. bounds that give better results if you know some nice properties of your data.

of w^* [2]. The goal of this problem is to show that when running the perceptron algorithm on S, the number of mistakes is bounded by $\left(\frac{\epsilon}{\gamma}\right)^2 + 1$ (which is arbitrarily better than the standard bound).

Let i_T be the index of the element on which the Tth mistake was made. Let w_T be the weight vector after T mistakes have been made. Note that $w_0 = 0$.

(ii) Show that $\|w_T\|^2 \le \epsilon^2 T + \sum_{t=1}^{T} \|Px_{i_t}\|^2$.
Hint: Start by showing that $\|w_T\|^2 \le \sum_{t=1}^{T} \|x_{i_t}\|^2$. Also, it may be helpful for both (ii) and (iii) to review the properties of projection matrices (but make sure you prove any facts you use).

(iii) Show that $\left(w_T \cdot w^*\right)^2 \ge T(T-1)\gamma^2 + \sum_{t=1}^{T} \|Px_{i_t}\|^2$.
Hint: start by showing that $w_T \cdot w^* = \sum_{t=1}^{T} y_{i_t} x_{i_t} \cdot w^*$.

(iv) Use parts (ii) and (iii) to show that $T \le \left(\frac{\epsilon}{\gamma}\right)^2 + 1$. Notice (for yourself, no writing necessary) that you successfully proved the tighter bound!

3 **[Learning DNFs with kernel perceptron]** Suppose that we have $S = \{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$ with $x^{(i)} \in \{0,1\}^d$ and $y^{(i)} \in \{-1,1\}$. Let $\varphi : \{0,1\}^d \to \{0,1\}$ be a "target function" which "labels" the points. Additionally assume that φ is a DNF formula (i.e. φ is a disjunction of conjunctions, or a boolean "or" of a bunch of boolean "and"s). The fact that it "labels" the points simply means that $\mathbf{1}[y^{(i)} = 1] = \varphi(x^{(i)})$.

For example, let $\varphi(x) = (x_1 \wedge x_2) \vee (x_1 \wedge \bar{x}_2 \wedge x_3)$ (where x_i denotes the ith entry of x), $x^{(i)} = \begin{pmatrix}1 & 0 & 1\end{pmatrix}^{\mathsf{T}}$, and $x^{(j)} = \begin{pmatrix}1 & 0 & 0\end{pmatrix}^{\mathsf{T}}$. Then, we would have $\varphi(x^{(i)}) = 1$ and $\varphi(x^{(j)}) = 0$, and thus $y^{(i)} = 1$ and $y^{(j)} = -1$.

(i) Give an example target function φ (make sure its a DNF formula) and set S such that the data is not linearly separable.

Part (i) clearly shows that running the perceptron algorithm on S cannot work in general since the data does not need to be linearly separable. However, we can try to use a feature transformation and the kernel trick to linearize the data and thus run the kernelized version of the perceptron algorithm on these datasets.

Consider the feature transformation $\phi : \{0,1\}^d \to \{0,1\}^{3^d}$ which maps a vector x to the vector of all the conjunctions of its entries or of their negations. So for example if $d = 2$ then $\phi(x) = \begin{pmatrix}1 & x_1 & x_2 & \bar{x}_1 & \bar{x}_2 & x_1 \wedge x_2 & x_1 \wedge \bar{x}_2 & \bar{x}_1 \wedge x_2 & \bar{x}_1 \wedge \bar{x}_2\end{pmatrix}^{\mathsf{T}}$ (note that 1 can be viewed as the empty conjunction, i.e. the conjunction of zero literals).

Let $K : \{0,1\}^d \times \{0,1\}^d \to \mathbb{R}$ be the kernel function associated with ϕ (i.e. for $a, b \in \{0,1\}^d : K(a,b) = \phi(a) \cdot \phi(b)$). Note that the naive approach of calculating $K(a,b)$ (simply calculating $\phi(a)$ and $\phi(b)$ and taking the dot product) takes time $\Theta(3^d)$.

Also let $w^* \in \{0,1\}^{3^d}$ be such that $w_1^* = -0.5$ (this is the entry which corresponds to the empty conjunction, i.e. $\forall x \in \{0,1\}^d : \phi(x)_1 = 1$) and $\forall i > 1 : w_i^* = 1$ iff the

[2] Geometrically the data resides in a high dimensional oval (ellipsoid) where the longest axis is in the direction of w^* and has radius R, and the other axes have radius ϵ. While this assumption seems quite construed (if we know that the longest axis is in the direction of w^* it seems that we could trivially find w^*), similar situations can be quite common when doing metric learning (which would approximately stretch the dataset in the direction of w^* while compressing in the orthogonal directions, hence resulting in a similarly oval shaped dataset).

ith conjunction is one of the conjunctions of φ. So for example in the above case where $d = 2$ and $\phi(x) = \begin{pmatrix}1 & x_1 & x_2 & \overline{x_1} & \overline{x_2} & x_1 \wedge x_2 & x_1 \wedge \overline{x_2} & \overline{x_1} \wedge x_2 & \overline{x_1} \wedge \overline{x_2}\end{pmatrix}^\mathsf{T}$ and letting $\varphi(x) = (x_1 \wedge x_2) \vee (\overline{x_1})$ we would have:

$$w^* = \begin{pmatrix}-0.5 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0\end{pmatrix}^\mathsf{T}$$

(ii) Find a way to compute $K(a, b)$ in $O(d)$ time.

(iii) Show that w^* linearly separates $\phi(S)$ ($\phi(S)$ is just a shorthand for $\{(\phi(x^{(i)}), y^{(i)})\}_{i=1}^n$) and find a lower bound for the margin γ with which it separates the data. Remember that $\gamma = \min_{(\phi(x^{(i)}),y^{(i)})\in\phi(S)} y_i \left(\frac{w^*}{\|w^*\|} \cdot \phi(x^{(i)})\right)$. Your lower bound should depend on s, the number of conjunctions in φ.

(iv) Find an upper bound on the radius R of the dataset $\phi(S)$. Remember that

$$R = \max_{(\phi(x^{(i)}),y^{(i)})\in\phi(S)} \|\phi(x^{(i)})\|.$$

(v) Use parts (ii), (iii), and (iv) to show that we can run kernel perceptron efficiently on this transformed space in which our data is linearly separable (show that each iteration takes $O(nd)$ time only) but that unfortunately the mistake bound is very bad (show that it is $O(s2^d)$).

There are ways to get a better mistake bound in this same kernel space, but the running time then becomes very bad (exponential). It is open whether there are ways to get both polynomial mistake bound and running time.

4 **[Inconsistency of the fairness definitions]** Recall from the previous homework the notation and definitions of group-based fairness conditions:

Notation:

Denote $X \in \mathbb{R}^d$, $A \in \{0, 1\}$ and $Y \in \{0, 1\}$ to be three random variables: non-sensitive features of an instance, the instance's sensitive feature and the target label of the instance respectively, such that $(X, A, Y) \sim \mathcal{D}$. Denote a classifier $f : \mathbb{R}^d \to \{0, 1\}$ and denote $\hat{Y} := f(X)$.

For simplicity, we also use the following abbreviations:

$$\mathbb{P} := \mathbb{P}_{(X,A,Y)\sim D} \qquad \text{and} \qquad \mathbb{P}_a := \mathbb{P}_{(X,a,Y)\sim D}$$

Group based fairness definitions:

- *Demographic Parity (DP)*

$$\mathbb{P}_0[\hat{Y} = \hat{y}] = \mathbb{P}_1[\hat{Y} = \hat{y}] \qquad \forall \hat{y} \in \{0, 1\}$$

(equal positive rate across the sensitive attribute)

- *Equalized Odds (EO)*

$$\mathbb{P}_0[\hat{Y} = \hat{y} \mid Y = y] = \mathbb{P}_1[\hat{Y} = \hat{y} \mid Y = y] \qquad \forall \hat{y},\ y \in \{0, 1\}$$

(equal true positive- and true negative-rates across the sensitive attribute)

- *Predictive Parity (PP)*

$$\mathbb{P}_0[Y = y \mid \hat{Y} = \hat{y}] = \mathbb{P}_1[Y = y \mid \hat{Y} = \hat{y}] \qquad \forall \hat{y},\ y \in \{0, 1\}$$

(equal positive predictive- and negative predictive-value across the sensitive attribute)

Unfortunately, achieving all three fairness conditions simultaneously is not possible. An impossibility theorem for group-based fairness is stated as follows.

– If A is dependent on Y, then Demographic Parity and Predictive Parity cannot hold at the same time.

– If A is dependent on Y and $\hat{Y}$ is dependent on Y, then Demographic Parity and Equalized Odds cannot hold at the same time.

– If A is dependent on Y, then Equalized Odds holds and Predictive Parity cannot hold at the same time.

These three results collectively show that it is impossible to simultaneously satisfy the fairness definitions except in some trivial cases.

(i) State a scenario where all three fairness definitions are satisfied simultaneously.

(ii) Prove the first statement.

(iii) Prove the second statement.

(iv) Prove the third statement.

Hint: First observe that

$$\mathbb{P}_0[Y = y|\hat{Y} = \hat{y}] = \mathbb{P}_1[Y = y|\hat{Y} = \hat{y}]\ \forall \hat{y},\ y \in \{0, 1\}$$

is equivalent to:

$$\mathbb{P}_0[Y = 1|\hat{Y} = \hat{y}] = \mathbb{P}_1[Y = 1|\hat{Y} = \hat{y}]\ \forall \hat{y} \in \{0, 1\}.$$

A necessary condition for PP is the equality of positive predictive value (PPV):

$$\mathbb{P}_0[Y = 1|\hat{Y} = 1] = \mathbb{P}_1[Y = 1|\hat{Y} = 1]$$

To prove the third statement, it is enough to prove a stronger statement: if A is dependent on Y, Equalized Odds and equality of Positive Predictive Value cannot hold at the same time.

Next, try to express the relationship between FPR$_a$ ($= \mathbb{P}_a[\hat{Y} = 1|Y = 0]$) and FNR$_a$ ($= \mathbb{P}_a[\hat{Y} = 0|Y = 1]$) using p_a ($= \mathbb{P}[A = a]$) and PPV$_a$ ($= \mathbb{P}_a[Y = 1|\hat{Y} = 1]$), $\forall a \in \{0, 1\}$ and finish the proof.

5 **[Neural Networks as Universal Function Approximators]** Neural networks are a flexible class of parametric models which form the basis for a wide range of algorithms in machine learning. Their widespread success is due both to the ease and efficiency of "training" them, or optimizing over their parameters, using the backpropagation algorithm, and their ability to approximate any smooth function. Recall that a feed-forward neural network is simply a combination of multiple 'neurons' such that the output of one neuron is fed as the input to another neuron. More precisely, a neuron ν_i is a computational unit that takes in an input vector $\mathbf{x}$ and returns a weighted combination of the inputs (plus the bias associated with neuron ν_i) passed through an activation function $\sigma : \mathbb{R} \to \mathbb{R}$, that is: $\nu_i(\mathbf{x}; \mathbf{w}_i, b_i) := \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i)$.

With this notation, we can define a layer ℓ of a neural network $\mathcal{N}^\ell$ that takes in an I-dimensional input and returns a O-dimensional output as

$$\begin{aligned} \mathcal{N}^\ell(x) &:= \big(\nu_1^\ell(\mathbf{x}), \nu_2^\ell(\mathbf{x}), \cdots, \nu_O^\ell(\mathbf{x})\big) & x \in \mathbb{R}^I \\ &= \sigma(W_\ell^{\mathsf{T}}\mathbf{x} + \mathbf{b}_\ell) & W_\ell \in \mathbb{R}^{d_I \times d_O} \text{ defined as } [\mathbf{w}_1^\ell, \ldots, \mathbf{w}_O^\ell], \\ && \text{and } \mathbf{b}_\ell \in \mathbb{R}^{d_O} \text{ defined as } [b_1^\ell, \ldots, b_O^\ell]^{\mathsf{T}}. \end{aligned}$$

Here $\mathbf{w}_i^\ell$ and b_i^ℓ refers to the weight and the bias associated with neuron ν_i^ℓ in layer ℓ, and the activation function σ is applied pointwise. An L-layer (feed-forward) neural network $\mathcal{F}_{L\text{-layer}}$ is then defined as a network consisting of network layers $\mathcal{N}^1, \ldots, \mathcal{N}^L$, where the input to layer i is the output of layer $i-1$. By convention, input to the first layer (layer 1) is the actual input data.

(i) Consider a nonlinear activation function $\sigma : \mathbb{R} \to \mathbb{R}$ defined as $x \mapsto 1/(1+e^{-x})$. Show that $\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1-\sigma(x))$.

(ii) Consider a *single layer* feed forward neural network that takes a d_I-dimensional input and returns a d_O-dimensional output, defined as $\mathcal{F}_{1\text{-layer}}(x) := \sigma(W^T x + b)$ for some $d_I \times d_O$ weight matrix W and a $d_O \times 1$ vector b. Given a training dataset $(x_1, y_1), \ldots, (x_n, y_n)$, we can define the *average error* (with respect to the network parameters) of predicting y_i from input example x_i as:

$$E(W, b) := \frac{1}{2n} \sum_{i=1}^{n} \|\mathcal{F}_{1\text{-layer}}(x_i) - y_i\|^2.$$

What is $\frac{\partial E}{\partial W}$, and $\frac{\partial E}{\partial b}$?

(note: we can use this gradient in a descent-type procedure to minimize this error and learn a good setting of the weight matrix that can predict y_i from x_i.)

(iii) Single layer neural networks—though reasonably expressive—are not flexible enough to approximate arbitrary smooth functions. Here we will focus on approximating some fun two-dimensional parametric functions $f : [0,1]^2 \to \mathbb{R}$ with a *multi-layer* neural network.

Consider an L-layer neural network $\mathcal{F}_{L\text{-layer}}$ as described above. Given a sample of input-output pairs $(x_1, y_1), \ldots, (x_n, y_n)$ generated from an unknown fixed function f, one can approximate f from $\mathcal{F}_{L\text{-layer}}$ by minimizing the following error function over the parameters W^T and b.

$$\begin{aligned} E(W_1, b_1, \ldots, W_L, b_L) &:= \frac{1}{2n} \sum_{i=1}^{n} \|\mathcal{F}_{L\text{-layer}}(x_i) - y_i\|^2 \\ &= \frac{1}{2n} \sum_{i=1}^{n} \|\mathcal{N}^L \circ \cdots \circ \mathcal{N}^1(x_i) - y_i\|^2. \end{aligned}$$

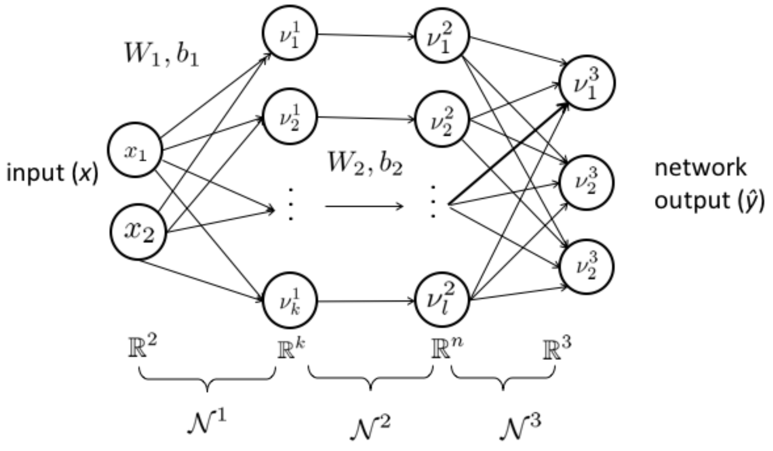
First we will explore how to train such a general network efficiently.

(a) At its core, a neural network is a chain of composed linear transformations connected by non-linearities, i.e. $\mathcal{N}(x) = \sigma_1 \circ f_1 \circ ... \circ \sigma_n \circ f_n$. Using the chain rule, compute the derivative of $\mathcal{N}(x)$. Naively, throwing away the result of each computation, what is the complexity of evaluating the derivative of a chain of n functions, in terms of the length of the chain n?

(b) Noticing that the computation of the derivative $\mathcal{N}'(x)$ involves computing $f_2 \circ ... \circ f_n(x)$, $f_3 \circ ... \circ f_n(x)$, and many other intermediate terms, describe how we can we improve the naive approach from the previous part by reusing computations performed when evaluating the function to compute the derivative at the same point. What is the complexity of this algorithm, again in terms of the length of the chain n? This is the *backpropagation algorithm.*

(iv) Now we will combine all of these calculations to implement a flexible framework for learning neural networks on arbitrary data. Your task is to implement a gradient descent procedure to learn the parameters of a general L-layer feed forward neural network using the backpropogation algorithm.

You will use this framework to learn to approximate complicated patterns in images. The provided data has a 2-dimensional input corresponding to the coordinate of a given pixel, i.e. $X = [(0,0), (0,1), ..., (n,m)]$, and the output is a one- or three-dimensional value giving a greyscale or RGB value for the image at that coordinate respectively. Note that since each x_i is 2-dimensional, layer $\mathcal{N}^1$ only contains two neurons. All other layers can contain an arbitrary number, say $k_1, ..., k_L$, neurons. The output layer will have either one or three neurons. Graphically the network you are implementing looks as follows, possibly with more intermediate layers.



Here is the pseudocode of your implementation:

```
Learn K-layer neural network for 2-d functions
input: data $(x_1, y_1), \ldots, (x_n, y_n)$,
.         size of the intermediate layers $k_1, ..., k_L$
- Initialize weight parameters $(W_1, b_1), ..., (W_L, b_L)$ randomly
- Repeat until convergence:
-   for a subset of training examples $\{(x_1, y_1), ..., (x_k, y_k)\}$
-       compute the network output $\hat{y}_i$ on $x_i$
-   compute gradients $\frac{\partial E}{\partial W_1}$, $\frac{\partial E}{\partial b_1}$, ..., $\frac{\partial E}{\partial W_L}$, $\frac{\partial E}{\partial b_L}$ using results saved from the forward pass
-       update weight parameters:
-           For all $i$, $W_i^{\text{new}} := W_i - \eta \frac{\partial E}{\partial W_i}$,    $b_i^{\text{new}} := b_i - \eta \frac{\partial E}{\partial b_i}$
```

You must submit your code on Courseworks to receive full credit.

Some hints: some example skeleton code `hw2_nnskeletoncode.py` has been provided (see Piazza HW2 post) as a possible way to structure your general neural network framework. You are not required to follow this skeleton code or to use Python. It merely provides inspiration for a possible way of structuring the project. This is close to how Tensorflow and PyTorch work.

– You should be able to use different numbers of intermediate layers and different size inputs and outputs to try and improve the convergence and performance of your algorithm.
– We were able to reconstruct both images with a 3-layer network and between 128 and 512 dimensional hidden layers. Training can take as little as 5 minutes with Numpy or Matlab. Minibatch size can be quite small (at least initially) to accelerate learning.
– Try using a simple function like a quadratic function to test your framework before attempting something more complicated like the image data provided.
– Try looking at how PyTorch or Tensorflow structure their computational graphs. Since this is a purely linear network, you can represent it as a linked-list of layers, each of which can pass the gradient to its child node.

Different gradient descent methods: Once we have computed the gradients, there are many ways to perform gradient descent (recall HW1 Q1) to optimize the function. **Stochastic Gradient Descent** (SGD) takes a subset of the data, averages the gradients over that subset, and steps iteratively in that direction. In this assignment, we are going to use **Adam** (short for Adaptive Momentum), a more sophisticated method that adapts the step size dynamically for each component of the gradient. The algorithm is:

Optimize a loss function with Adam

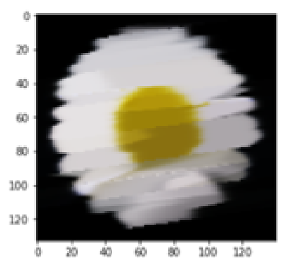
```
*input:* data $(x_1, y_1), \ldots, (x_n, y_n)$,
- Initialize $\beta_1 = 0.9$, $\beta_2 = 0.999$
- For all $i$, initialize $m_i = \mathbf{0}$, $v_i = \mathbf{0}$, zero vectors with same shape as weights $W_i$.
- Repeat until convergence:
-   for a subset of training examples $\{(x_1, y_1), ..., (x_k, y_k)\}$
-       compute the network output $\hat{y}_i$ on $x_i$
-   compute gradients $\frac{\partial E}{\partial W_1}, \frac{\partial E}{\partial b_1}, ..., \frac{\partial E}{\partial W_L}, \frac{\partial E}{\partial b_L}$
-       update weight parameters (do the same for bias terms):
-           For all $i$, $m_i^{\text{new}} := (\beta_1)m_i - (1 - \beta_1)\frac{\partial E}{\partial W_i}$, $v_i^{\text{new}} := (\beta_2)v_i - (1 - \beta_2)\left(\frac{\partial E}{\partial W_i}\right)^2$
-           For all $i$, $W_i^{\text{new}} := W_i - \eta\frac{m_i}{\sqrt{v_i} + \epsilon}$
```

Do the same for the biases as well. For more information, look at `http://cs231n.github.io/neural-networks-3/#ada`. This gradient descent algorithm will allow your network to learn more flexibly. Start by implementing traditional gradient descent, and then add Adam after the network works on simple inputs.

(v) Download `hw2_data.mat` (see Piazza HW2 post). It contains data for two distinct images, each with vector valued variables X and Y (they will be labeled $X1$, $X2$, and $Y1$ and $Y2$ for the two images). Each row in Y gives the pixel value at the corresponding coordinate in X.

You can visualize the data by reformatting the image data Y into a 2D array and displaying it as an image (`imshow` in Python or Matlab may be useful). For example, the second image (and our reconstruction using a very simple network) looks like this: Use

(a) original image


(b) reconstructed

your implementation to learn the unknown mapping function which can yield Y from X for both image datasets. Once the network parameters are learned, plot the generated pixel values as an image and include the results with your writeup. Compare the results for at least two settings for the size and number of hidden layers in your network.