Introduction to Machine Learning

Neural Networks

Due: Thursday, March 21, 2024 at 11:00 PM

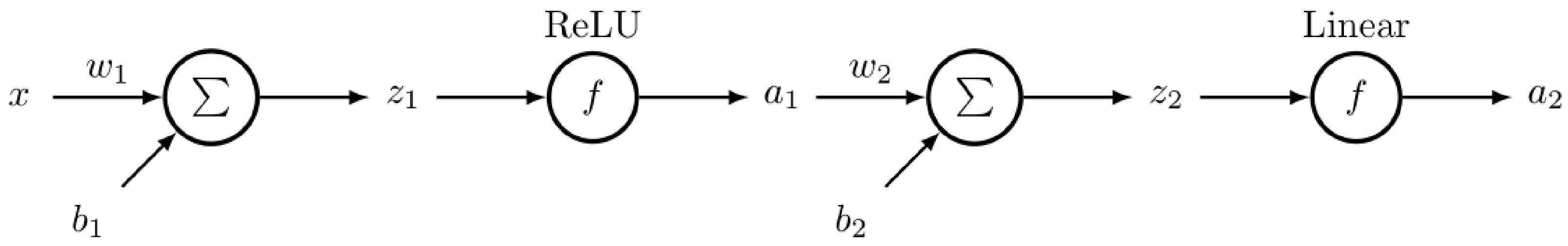
This homework builds on the material in the course notes on Neural Networks.

1) Simple Neural Network

Here is a very simple neural network with one hidden unit (ReLU) and one linear output unit. The weights and biases for this network are given by

$$w_1 = -2, b_1 = 3, w_2 = 3, b_2 = 0$$

Let a_1 and a_2 denote the activations of the hidden and output unit, respectively, and $z_1 = w_1 x + b_1$ and $z_2 = w_2 a_1 + b_2$ the pre-activation inputs to these units, where $a_1 = \text{ReLU}(z_1)$ and $a_2 = z_2$.



1.1)

What is the numerical value of the output unit, i.e., the value of a_2 when presented with an input $x = 1$?

Enter a number: 3

Check Formatting | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.2)

We assume that the loss is simply the squared loss, i.e. $\text{Loss}(g, y) = (g - y)^2$. Give an expression for $\frac{\partial L(z_2, y)}{\partial z_2}$, the gradient of the loss with respect to z_2. Your final answer should no longer include derivative signs, and should be in terms of `x`, `y`, `w_1`, `w_2`, `b_1`, and `b_2`. You can also use `max(a, b)` for the $\max\{a, b\}$ function.

Please use `zero` to indicate either a zero vector, or the scalar `0`.

Enter an expression for $\frac{\partial L(z_2, y)}{\partial z_2}$: 2 * (w_2 * max((w_1 * x + b_1), zero) + b_2 - y)

Check Syntax | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.3)

Under the same assumption of squared loss, give an expression for the gradient of the loss with respect to the input to the hidden unit, i.e., with respect to z_1. Your final answer should no longer include derivative signs, and should be in terms of `x`, `y`, `w_1`, `w_2`, `b_1`, and `b_2`. Remember to use the chain rule.

Enter an expression for $\frac{\partial L(z_2,y)}{\partial z_1}$ when $w_1 x + b_1 < 0$: 0

Check Syntax | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

Enter an expression for $\frac{\partial L(z_2,y)}{\partial z_1}$ when $w_1 x + b_1 \geq 0$: 2 * (w_2 * (w_1 * x + b_1) + b_2 - y) * w_2

Check Syntax | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.4)

Let input $x = 1$ and target output $y = 1$. We continue to use the squared loss. Provide the numerical value of w_1 after a single SGD update with learning rate 0.5, under the assumption that, $w_1 = -2, b_1 = 3, w_2 = 3, b_2 = 0$.

Enter the value of w_1 after one step of SGD: 2 * (2) * -2

Check Syntax | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

1.5)

Under what conditions will w_1 be unchanged by a back-propagation update step given training example (x, y)?

Choose all that apply:

- [✓] $x = 0$
- [ ] $y = 0$
- [ ] $w_1 = 0$
- [✓] $w_2 = 0$
- [✓] $w_1 x + b_1 < 0$
- [✓] $z_2 = y$

Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

2) Neural Networks

In this problem we will analyze a simple neural network to understand its classification properties. Here is a reminder about our notation for general neural networks. Each layer has multiple inputs and outputs, and can be broken down into two parts:

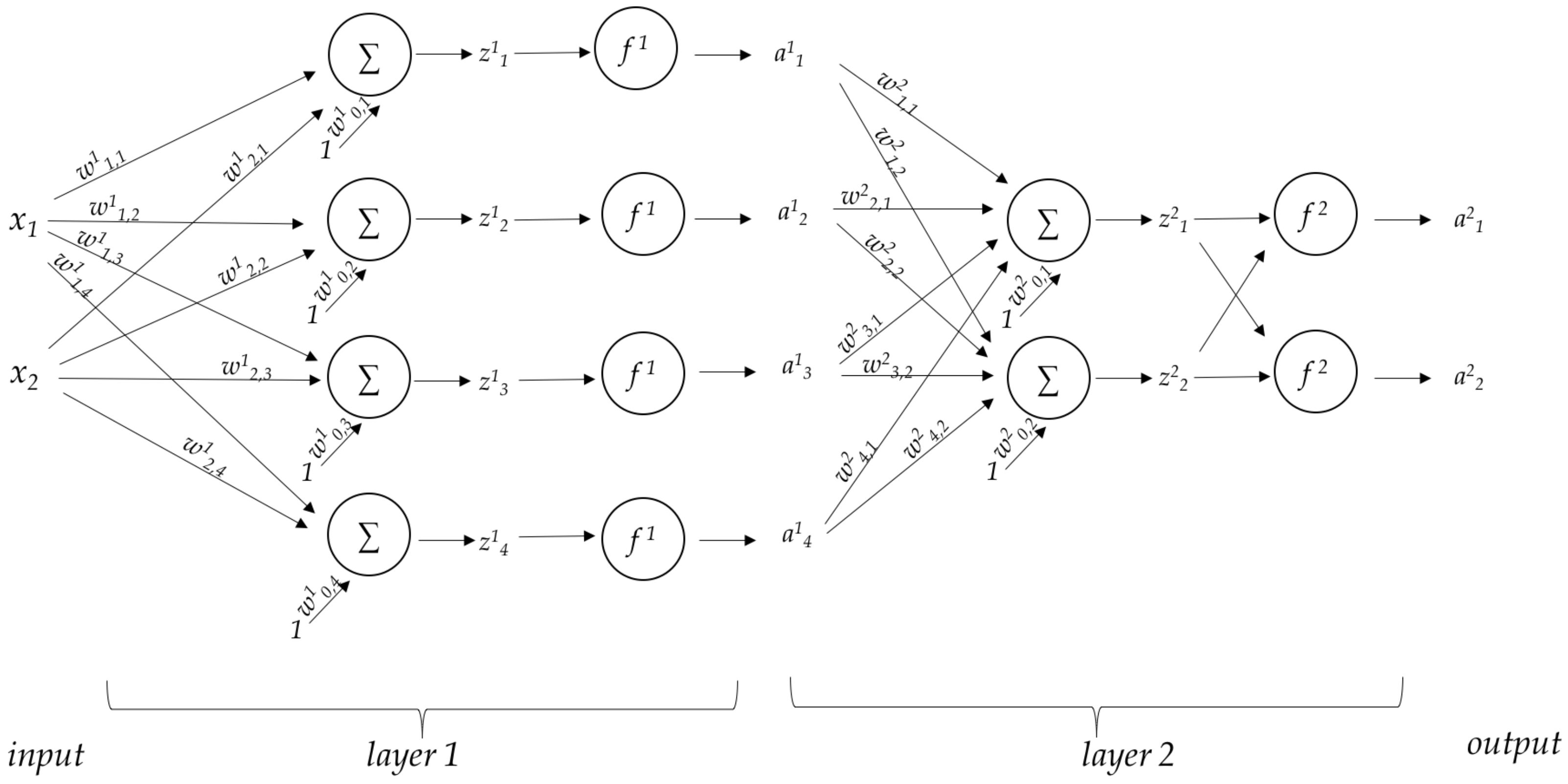
- A **linear** module that implements a linear transformation: $z_j = (\sum_{i=1}^{m} x_i w_{i,j}) + w_{0\,j}$ specified by a weight matrix W and a bias vector W_0. The output is $[z_1, \ldots, z_n]^T$.
- An **activation** module that applies an activation function to the outputs of the linear module for some activation function f, such as tanh or ReLU in the hidden layers or softmax (see below) at the output layer. We write the output as: : $[f(z_1), \ldots, f(z_m)]^T$, although technically, for some activation functions such as softmax, each output will depend on all the z_i, not just one.

We will use the following notation for quantities in a network:

- Inputs to the network are $x_1, \ldots, x_d$.
- Number of layers is L
- The weight matrix for layer l is W^l, an $m^l \times n^l$ matrix, and the bias vector (offset) is W_0^l, an $n^l \times 1$ vector
- There are m^l inputs to layer l

- There are $n^l = m^{l+1}$ outputs from layer l
- The outputs of the linear module for layer l are known as **pre-activation** values and denoted z^l
- The activation function at layer l is $f^l(\cdot)$
- Layer l activations are $a^l = [f^l(z_1^l), \ldots, f^l(z_{n^l}^l)]^T$
- The output of the network is the values $a^L = [f^L(z_1^L), \ldots, f^L(z_{n^L}^L)]^T$
- Loss function $Loss(a, y)$ measures the loss of output values a when the target is y
- There are b **data points** (we've seen the notation as n datapoints before, but now n is used for number of layer outputs)

Consider the neural network given in the figure below, with ReLU activation functions (f^1 in the figure) on all hidden neurons, and softmax activation (f^2 in the figure) for the output layer, resulting in softmax outputs (a_1^2 and a_2^2 in the figure).



Given an input $x = [x_1, x_2]^T$, the hidden units in the network are activated in stages as described by the following equations:

$$
\begin{aligned}
z_1^1 &= x_1 w_{1,1}^1 + x_2 w_{2,1}^1 + w_{0,1}^1 & a_1^1 &= \max\{z_1^1, 0\} \\
z_2^1 &= x_1 w_{1,2}^1 + x_2 w_{2,2}^1 + w_{0,2}^1 & a_2^1 &= \max\{z_2^1, 0\} \\
z_3^1 &= x_1 w_{1,3}^1 + x_2 w_{2,3}^1 + w_{0,3}^1 & a_3^1 &= \max\{z_3^1, 0\} \\
z_4^1 &= x_1 w_{1,4}^1 + x_2 w_{2,4}^1 + w_{0,4}^1 & a_4^1 &= \max\{z_4^1, 0\}
\end{aligned}
$$

$$
\begin{aligned}
z_1^2 &= a_1^1 w_{1,1}^2 + a_2^1 w_{2,1}^2 + a_3^1 w_{3,1}^2 + a_4^1 w_{4,1}^2 + w_{0,1}^2 \\
z_2^2 &= a_1^1 w_{1,2}^2 + a_2^1 w_{2,2}^2 + a_3^1 w_{3,2}^2 + a_4^1 w_{4,2}^2 + w_{0,2}^2.
\end{aligned}
$$

The final output of the network is obtained by applying the *softmax* function to the last hidden layer,

$$
\begin{aligned}
a_1^2 &= \frac{e^{z_1^2}}{e^{z_1^2} + e^{z_2^2}} \\
a_2^2 &= \frac{e^{z_2^2}}{e^{z_1^2} + e^{z_2^2}}.
\end{aligned}
$$

In this problem, we will consider the following setting of parameters:

$$\begin{bmatrix} w^1_{1,1} & w^1_{1,2} & w^1_{1,3} & w^1_{1,4} \\ w^1_{2,1} & w^1_{2,2} & w^1_{2,3} & w^1_{2,4} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}, \qquad \begin{bmatrix} w^1_{0,1} \\ w^1_{0,2} \\ w^1_{0,3} \\ w^1_{0,4} \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \end{bmatrix}$$

$$\begin{bmatrix} w^2_{1,1} & w^2_{1,2} \\ w^2_{2,1} & w^2_{2,2} \\ w^2_{3,1} & w^2_{3,2} \\ w^2_{4,1} & w^2_{4,2} \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 1 & -1 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}, \qquad \begin{bmatrix} w^2_{0,1} \\ w^2_{0,2} \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \end{bmatrix}.$$

2.1) ReLU activation

2.1.1)

Write a short program to compute the derivative of the ReLU (f^1) with respect to any z^l_j. As you might have realized, ReLU is not differentiable at 0. However, for this exercise, you can assume that the derivative at 0 is 0.

- `z` is a column vector

It should return a column vector. Hint: `np.where` might be a useful function.

```python
# We recommend looking at the ReLU plot for better understanding.
def dReLU_dz(z):
    d = [0 if i<=0 else 1 for i in z]
    return np.array([d]).T

```

Run Code | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

2.2) Output

Consider the input $x_1 = 3$, $x_2 = 14$.

2.2.1)

What are the outputs of the hidden units, $[a^1_1, a^1_2, a^1_3, a^1_4]$?

Enter a Python list of four numbers: [2, 13, 0, 0]

Check Formatting | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

2.2.2)

What is the final output (a^2_1, a^2_2) of the network?

Enter a Python list of two numbers: [1, 0]

Check Formatting | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

2.3) Unit decision boundaries

Let's characterize the *decision boundaries* in x-space, corresponding to the four hidden units. These are the regions where the input to the units $z_1^1, z_2^1, z_3^1, z_4^1$ are exactly zero.

Hint: You should draw a diagram of the decision boundaries for each unit in the x-space and label the sides of the boundaries with 0 and $+$ to indicate whether the unit's output would be exactly 0 or positive, respectively. (The diagram should be a 2D plot with x_1 and x_2 on each axis, with lines for $z_1^1 = 0$, $z_2^1 = 0$, $z_3^1 = 0$, $z_4^1 = 0$.)

2.3.1)

What is the shape of the decision boundary for a single unit?

Choose one: Line

Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

2.3.2)

Enter 4 different points that lie on the decision boundary of the first hidden unit. For each of these points, the value of the first hidden unit, z_1^1 (the first element of z_1) should be 0. Enter these points as a 2 x 4 matrix. Note that there are many possible correct answers.

Enter a Python list of lists where each list is a row of the matrix: [[1, 1, 1, 1], [0, 1, 2, 3]]

Check Formatting | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

2.3.3)

Consider the following input vectors: $x^{(1)} = [0.5, 0.5]^T, x^{(2)} = [0, 2]^T, x^{(3)} = [-3, 0.5]^T$. Enter a matrix where each column represents the outputs of the hidden units ($a_1^1, \ldots, a_4^1$) for each of the input vectors. Following 2.3.2's response style, this is a 4 x 3 matrix.

Hint: Make a diagram of the decision boundaries of all the units.

Enter a Python list of lists where each list is a row of the matrix: [[0, 0, 0], [0, 1, 0], [0, 0, 2], [0, 0, 0]]

Check Formatting | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

2.4) Network outputs

In our network above, the output layer with two softmax units is used to classify into one of two classes. For class 1, the first unit's output should be larger than the other unit's output, and for class 2, the second unit's output should be larger. This generalizes nicely to k classes by using k output units.

We have previously examined addressing two-class classification problems using a single output unit with a sigmoid activation. Using softmax allows us to generalize classification to multiple classes.

Let's characterize the region in x-space where this network's output indicates the first class (that is, a_1^2 is larger than a_2^2) or indicates the second class (that is, a_2^2 is larger than a_1^2). Your diagram from the previous part will be useful here.

What is the output value of the neural network in each of the following cases? Write your answer for a_i^2 as expressions, you can use powers of e, for example, `e**2 + 1`; the exponents can be negative, `e**(-2) + 1`.

Case 1) For $f(z_1^1) + f(z_2^1) + f(z_3^1) + f(z_4^1) = 0$

2.4.1)

$a_1^2 =$ 1/2

Check Syntax | Submit | View Answer | Ask for Help 0.00%

You have infinitely many submissions remaining.

2.4.2)

$a_2^2 =$ e**2/(1+e**2)

Check Syntax | Submit | View Answer 100.00%

You have infinitely many submissions remaining.

2.4.3)

Which class is predicted? Class 2

Submit | View Answer 100.00%

You have infinitely many submissions remaining.

Case 2) For $f(z_1^1) + f(z_2^1) + f(z_3^1) + f(z_4^1) = 1$

2.4.4)

$a_1^2 =$ 1/2

Check Syntax | Submit | View Answer 100.00%

You have infinitely many submissions remaining.

2.4.5)

$a_2^2 =$ 1/2

Check Syntax | Submit | View Answer 100.00%

You have infinitely many submissions remaining.

2.4.6)

Which class is predicted? Boundary

Submit | View Answer 100.00%

You have infinitely many submissions remaining.

Case 3) For $f(z_1^1) + f(z_2^1) + f(z_3^1) + f(z_4^1) = 3$

2.4.7)

$a_1^2 =$ e**3/(e**3+e**-1)

Check Syntax | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

2.4.8)

$a_2^2 =$ e**-1/(e**3+e**-1)

Check Syntax | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

2.4.9)

Which class is predicted? Class 1

Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

3) Classification with Neural Networks

Mira's father is an archaeologist who appraises Chinese antiques. Since his daughter recently took 6.390, he asked her a favor: to build a classifier to predict from which dynasty each antique artifact originates. Specifically, each antique artifact was built by one of the four dynasties: Tang (A.D. 618-907), Song (A.D. 960-1276), Ming (A.D. 1368–1644), Qing (A.D. 1636-1912). Mira decides to build a classifier using a neural network and train it using multiclass negative log likelihood (NLLM) loss. Recall that the multiclass negative log likelihood loss for a single example is defined as:

$$L_{\text{NLLM}}(g, y) = -\sum_{i=1}^{n_y} y_i \log g_i$$

where $g = (g_1, \ldots, g_{n_y})$ denotes the predicted probability distribution over the classes and $y = (y_1, \ldots, y_{n_y})$ is the ground truth, a one hot vector that has zero at each index except at the correct class: $y = (1, 0, 0, 0)$ for Tang, $y = (0, 1, 0, 0)$ for Song, $y = (0, 0, 1, 0)$ for Ming, $y = (0, 0, 0, 1)$ for Qing.

3.1)

Assume that Mira is given an antique that belongs to Ming dynasty ($y = (0, 0, 1, 0)$). Which of these predictions has the smallest NLLM loss?

- ○ $g = (0.25, 0.20, 0.30, 0.25)$
- [✓ $g = (0.01, 0.01, 0.44, 0.54)$]
- ○ $g = (0.25, 0.25, 0.25, 0.25)$
- ○ $g = (0.97, 0.01, 0.01, 0.01)$

Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

3.2)

Apart from the NLLM loss, Mira is also thinking about trying out other loss functions. In particular, she is thinking about using the accuracy:

$$L_{\text{accuracy}}(g, y) = \begin{cases} -1 & \arg\max(y) = \arg\max(g) \\ 0 & \textit{otherwise} \end{cases}$$

or the squared loss function:

$$L_{\text{squared}}(g, y) = \left(1 - g y^{\top}\right)^2$$

as her new loss functions. Which of the these loss functions can be minimized by the stochastic gradient descent (SGD) algorithm (mark all that apply)?

- [✓] NLLM-loss, L_{NLLM}
- [ ] Accuracy, L_{accuracy}
- [✓] Squared loss, L_{squared}

Submit View Answer **100.00%**

You have infinitely many submissions remaining.

3.3)

After trying out different model architectures, Mira finds that the softmax classifier works well. When she uses softmax, the last layer of her network computes pre-activations $z = \left(z_1, \ldots, z_{n_y}\right)$ which may be arbitrarily large or small (Note: a pre-activation is the linearly weighted sum that is an input to the activation function). The softmax function then computes $g = \left(g_1, \ldots, g_{n_y}\right)$ by normalizing z such that the sum of the g_i is 1:

$$g_i = \frac{e^{z_i}}{\sum_{j=1}^{n_y} e^{z_j}}$$

Mira finds that for some settings of the pre-activation values, the basic softmax function works poorly. She finds out that subtracting the maximum preactivation value, $\max(z)$, from all individual pre-activations z_i produces more reliable results. Explain why.

Choose all that apply:

- [ ] This transformation affects the resulting values of $\hat{y}_i$ but it is still useful.
- [ ] It is more efficient to do this computation with numbers of smaller magnitude.
- [✓] Dividing a very large floating point number by another very large floating point number may generate imprecise results.

Submit View Answer **100.00%**

You have infinitely many submissions remaining.

3.4)

Say that Mira wanted to solve a slightly different problem: given an artifact, Mira would like to figure out what the probability is that the artifact is “typical” of each of the four time periods. That is, there could be an antique crafted in a style which was popular both during the Tang and Ming eras, but not at all in the other two eras, in which case ideally we would output $y = (1, 0, 1, 0)$. Choose a different structure (activation function and number of nodes) for the last layer. Specify a loss function that would work better for this multi-class labeling task.

Good possible choices of activation function and loss function are:

- [ ] Softmax and multi-class NLLM
- [✓] Four independent sigmoids and mean of NLL
- [ ] No output non-linearity and mean MSE over all four outputs
- [ ] Four independent sigmoids and multiclass NLLM

Submit View Answer **100.00%**

You have infinitely many submissions remaining.

4) Backpropagation

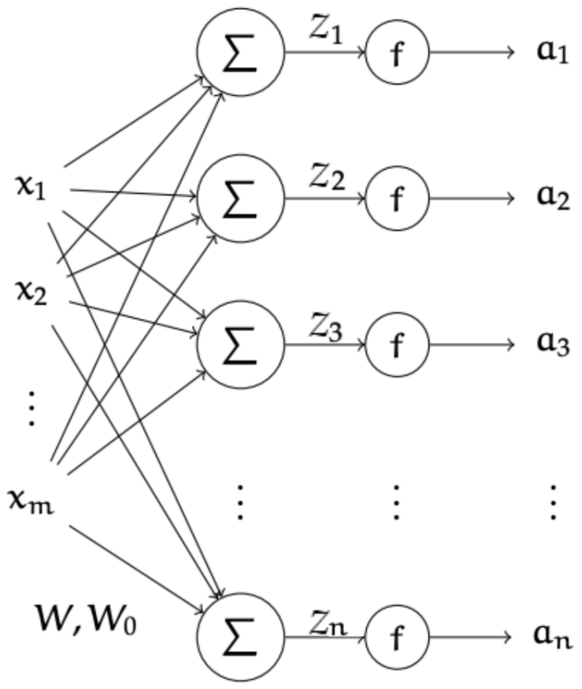
Now, we develop a general approach to specifying an L-layer neural networks, both the **forward** pass that takes an input x and produces a final output, and the **backward** pass that computes gradients of a loss function relative to the weights. We do this by treating the network as being made up of multiple **layers**, each of which can be broken down into two modules:

- A **linear** module that implements a linear transformation: $z_j = (\sum_{i=1}^{m} x_i w_{i,j}) + w_{0j}$ specified by a weight matrix W and a bias vector W_0. The output is $[z_1, \ldots, z_n]^T$.
- An **activation** module that applies an activation function to the outputs of the linear module for some activation function f, such as tanh or ReLU in the hidden layers or softmax (see below) at the output layer. We write the output as: : $[f(z_1), \ldots, f(z_m)]^T$, although technically, for some activation functions such as softmax, each output will depend on all the z_i, not just one.

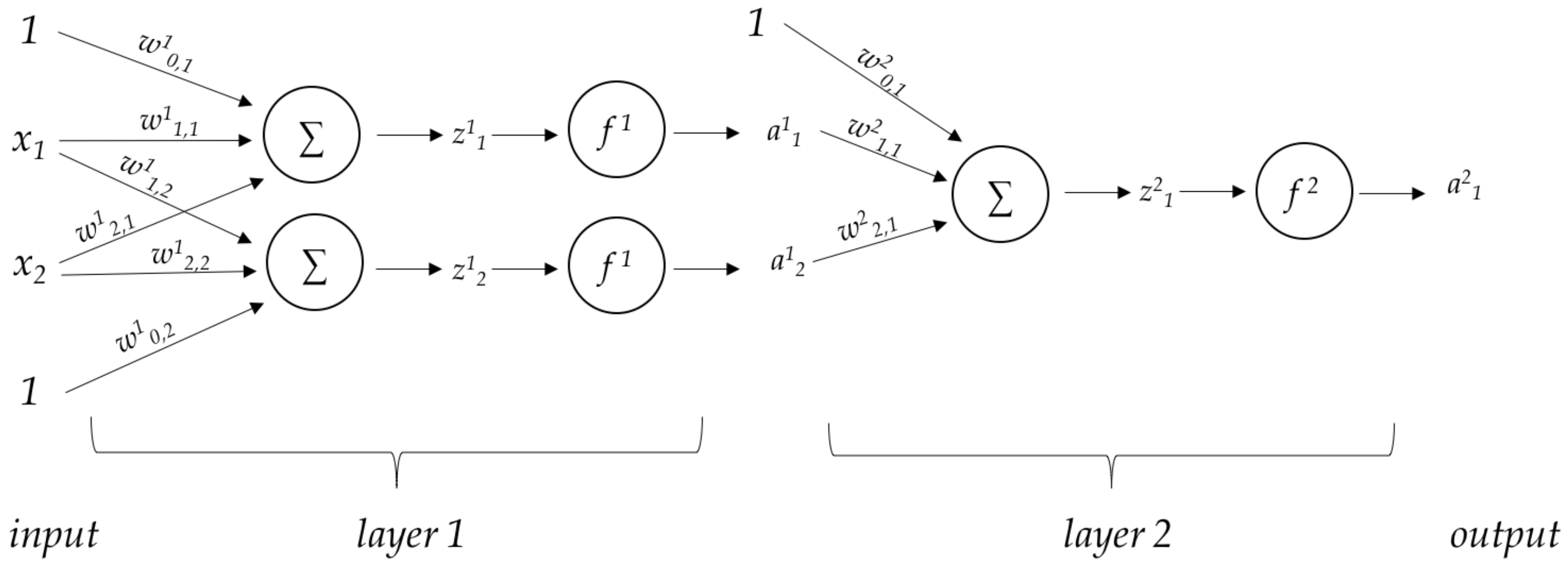
We will use the following notation for quantities in a network:

- Inputs to the network are $x_1, \ldots, x_d$.
- Number of layers is L
- The weight matrix for layer l is W^l, an $m^l \times n^l$ matrix, and the bias vector (offset) is W_0^l, an $n^l \times 1$ vector
- There are m^l inputs to layer l
- There are $n^l = m^{l+1}$ outputs from layer l
- The outputs of the linear module for layer l are known as **pre-activation** values and denoted z^l
- The activation function at layer l is $f^l(\cdot)$
- Layer l activations are $a^l = [f^l(z_1^l), \ldots, f^l(z_{n^l}^l)]^T$
- The output of the network is the values $a^L = [f^L(z_1^L), \ldots, f^L(z_{n^L}^L)]^T$
- Loss function $Loss(a, y)$ measures the loss of output values a when the target is y

Here is an illustrative picture of just a single layer for you to visualize while thinking about dimensions. Go through the above bullets and make sure you understand each set of notations and/or dimensions. Why should $n^l = m^{l+1}$? Given the dimensions for the input and output of a layer, can you see why W^l has the dimensions that it does? If these questions don't make sense, ask for help on Piazza or in office hours!



Here is another illustrative picture that illustrates multiple layers:



- Each linear module has a `forward` method that takes in a column vector of activations A (from the previous layer) and returns a column vector Z of pre-activations; it also stores its input or output vectors for use by other methods (e.g., for subsequent backpropagation).
- Each activation module has a `forward` method that takes in a column vector of pre-activations Z and returns a column vector A of activations; it also stores its input or output vectors for use by other methods (e.g., for subsequent backpropagation).
- Each linear module has a `backward` method that takes in a column vector $\frac{\partial Loss}{\partial Z}$ and returns a column vector $\frac{\partial Loss}{\partial A}$. This module also computes and stores $\frac{\partial Loss}{\partial W}$ and $\frac{\partial Loss}{\partial W_0}$, the gradients with respect to the weights.
- Each activation module has a `backward` method that takes in a column vector $\frac{\partial Loss}{\partial A}$ and returns a column vector $\frac{\partial Loss}{\partial Z}$.

The backpropagation algorithm will consist of:

- Calling the `forward` method of each module in turn, feeding the output of one module as the input to the next; starting with the input values of the network. After this pass, we have a predicted value for the final network output.
- Calling the `backward` method of each module in reverse order, using the returned value from one module as the input value of the previous one. The starting value for the backward method is $\partial Loss(a^L, y)/\partial a^L$, where a^L is the activation of the final layer (computed during the forward pass) and y is the desired output (the label).

4.1) Linear Module

The `forward` method, given A from the previous layer, implements:

$$Z = W^T A + W_0$$

and stores the input $m^l \times 1$ input A to be used by the `backward` method (please refer to the glossary above for the definitions of m^l and other variables).

The following questions ask for a matrix expression involving any of `A`, `Z`, `dLdA`, `dLdZ`,`W` and `W_0`.

** Enter your answers as Python expressions. You can use `transpose(x)` for transpose of an array, and `x@y` to indicate a matrix product of two arrays. Remember that `x*y` denotes component-wise multiplication.**

The `backward` method, given $\mathtt{dLdZ} = \frac{\partial Loss}{\partial Z}$ ($n^l \times 1$ vector), returns $\mathtt{dLdA} = \frac{\partial Loss}{\partial A}$ ($m^l \times 1$ vector), which is passed to the previous activation module in the neural network:

$$\frac{\partial Loss}{\partial A} = \frac{\partial Z}{\partial A}\frac{\partial Loss}{\partial Z}$$

4.1.1)

`dLdA` = W@dLdZ

Check Syntax | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

The `backward` method, given $\texttt{dLdZ} = \frac{\partial Loss}{\partial Z}$, also computes `dLdW` ($m^l \times n^l$ matrix) and stores it in the module instance.

$$\texttt{dLdW} = \frac{\partial Loss}{\partial W} = \frac{\partial Z}{\partial W}\frac{\partial Loss}{\partial Z}$$

4.1.2)

`dLdW` = A@transpose(dLdZ)

Check Syntax | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

The last part of the `backward` method is to compute and store `dLdW0` (an $n^l \times 1$ vector).

$$\texttt{dLdW0} = \frac{\partial Loss}{\partial W_0} = \frac{\partial Z}{\partial W_0}\frac{\partial Loss}{\partial Z}$$

4.1.3)

`dLdW0` = dLdZ

Check Syntax | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

We will use `dLdW` and `dLdW0` as the gradient values in SGD.

4.2) Activation Module

Activation modules don't have any weights to update and so they are simpler.

The `forward` method for functions like *tanh* or *sigmoid*, given Z, return the function on the vector, **component wise**. *Softmax* operates on the whole vector, as described in homework 4, and will need some special treatment.

The `backward` method, given $\texttt{dLdA} = \frac{\partial Loss}{\partial A}$, returns:

$$dLdZ = \frac{\partial Loss}{\partial Z} = \frac{\partial Loss}{\partial A}\frac{\partial A}{\partial Z}$$

In this case, $m^l = n^l$ and the quantities are column vectors of that size.

For softmax $= SM(Z)$ at the output layer and assuming that we are using *NLL* as the $Loss(A, Y)$ function, we have seen that there is a simple form for $\texttt{dLdZ} = \frac{\partial Loss}{\partial Z}$; namely, it is the prediction error $A - Y$. A similar result holds when using *NLL* with a sigmoid output activation or a quadratic loss with a linear output activation.

No question here, but please read as this information will be helpful!

5) Implementing Neural Networks

In this part of the homework, we will implement a neural network. You can use the colab notebook which contains the code for this problem.

You may wish to review the course notes on Gradient Descent, in addition to the notes on Neural Networks.

Although for "real" applications you want to use one of the many packaged implementations of neural networks (we'll start using one of those soon), there is no substitute for implementing one yourself to get an in-depth understanding. Luckily, that is relatively easy to do if we're not too concerned with maximum efficiency.

We'll use the modular implementation that we guided you through in the previous problem, which leads to clean code. The basic framework for GD training is given below. We can construct a network and train it as follows:

```
# build a 3-layer network
net = Sequential([Linear(2,3), Tanh(),
                  Linear(3,3), Tanh(),
                  Linear(3,2), SoftMax()], NLL())
# train the network on data and labels
net.gd(X, Y)
```

We call this a Sequential object because we feed the outputs of the previous layer into the next, you could say this forms a sequential relationship between layers

We will (later) be generalizing GD to operate on a "mini-batch" of data points instead of a single point. You should strive for an implementation of the forward, backward, and class_fun methods that works with batches of data. Note that when b is mentioned as part of the shape of a matrix in the code, this b refers to the number of data points.

You will need to fill in the missing code to create a sequential neural network model. We encourage you to test in your own Python environment and then paste your answer and verify the results. The test cases are provided in the code distribution or the colab file linked above in this page. **The code distribution includes the tests being run by the code checkers and additional test methods that will test each of the methods in turn, so you can debug incrementally.**

5.1) Linear Modules

Each linear module has a forward method that takes in a batch of activations A (from the previous layer) and returns a batch of pre-activations Z.

Each linear module has a backward method that takes in `dLdZ` and returns `dLdA`. This module also computes and stores `dLdW` and `dLdW0`, the gradients with respect to the weights.

Hint: Be careful with dimensions when computing `dLdW0`. `dLdZ` is $(n^l \times b)$, but `dLdW0` is $(n^l \times 1)$. DO NOT FORGET to sum over all b data points in the batch when computing `dLdW0`.

```python
class Module:
    def sgd_step(self, lrate): pass  # For modules w/o weights

class Linear(Module):
    def __init__(self, m, n):
        # initializes the weights randomly and offsets as 0
        self.m, self.n = (m, n)  # (in size, out size)
        self.W0 = np.zeros([self.n, 1])  # (n x 1)
        self.W = np.random.normal(0, 1.0 * m ** (-.5), [m, n])  # (m x n)

    def forward(self, A):
        # store the input matrix for future use
        self.A = A   # (m x b)  Hint: make sure you understand what b stands for
        return self.W.T@A + self.W0

    def backward(self, dLdZ):
        # dLdZ is (n x b), uses stored self.A
        # store the derivatives for use in sgd_step and returd dLdA
        self.dLdW = self.A@dLdZ.T    # Your code
        self.dLdW0 = dLdZ.sum(axis=1).reshape((self.n, 1))    # Your code
        return self.W @ dLdZ          # Your code: return dLdA (m x b)

    def sgd_step(self, lrate):  # Gradient descent step
        self.W -= lrate*(self.dLdW)          # Your code
        self.W0 -= lrate*(self.dLdW0)          # Your code

```

Run Code | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

5.2) Activation Modules

Each activation module has a forward method that takes in a batch of pre-activations Z and returns a batch of activations A. (Ask yourself, why is this the reverse of the linear module?)

Each activation module has a backward method that takes in **`dLdA`** and returns **`dLdZ`**, with the exception of softmax, where we assume **`dLdZ`** is passed in. (Ask yourself, why don't the activation modules need sgd_step methods?)

5.2.1) Tanh

Please use `np.tanh` here.

Hint: the derivative of $\tanh$ is given by $\frac{\partial \tanh(z)}{\partial z} = 1 - \tanh(z)^2$. If you're having problems with this, try writing out the shapes of everything involved and remember that `@` does matrix multiplication while $*$ does component-wise multiplication.

```python
class Tanh(Module):             # Layer activation
    def forward(self, Z):
        self.A = np.tanh(Z)            # Your code
        return self.A

    def backward(self, dLdA):    # Uses stored self.A
        return dLdA * (1 - np.tanh(np.arctanh(self.A))**2)             # Your code: return dLdZ

```

100.00%

You have infinitely many submissions remaining.

Here is the solution we wrote:

```python
class Tanh(Module):             # Layer activation
    def forward(self, Z):
        self.A = np.tanh(Z)
        return self.A

    def backward(self, dLdA):    # Uses stored self.A
        return dLdA * (1.0 - (self.A ** 2))
```

5.2.2) ReLU

Hint: `np.maximum` might be useful.

```python
class ReLU(Module):              # Layer activation
    def forward(self, Z):
        self.A = np.maximum(0, Z)           # Your code
        return self.A

    def backward(self, dLdA):    # uses stored self.
        return dLdA*np.zeros([1,self.A.shape[1]]) if np.sum(self.A) == 0 else dLdA*np.ones([1,self.
                 # Your code: return dLdZ

```

100.00%

You have infinitely many submissions remaining.

Here is the solution we wrote:

```python
class ReLU(Module):              # Layer activation
    def forward(self, Z):
        self.A = np.maximum(0, Z)
        return self.A

    def backward(self, dLdA):    # uses stored self.A
        return dLdA * (self.A != 0)
```

5.2.3) Softmax

We will now implement the softmax activation function. Note that we will assume that the derivative `dLdZ` is passed into the backward method. We also have the additional `class_fun` method:

- `SoftMax.class_fun`: Given `Ypred`, the column vector of class probabilities for each point (computed by softmax), this returns a vector of the classes (integers) with the highest probability for each point. The vector should be 1-dimensional with shape `(b,)`.

```
class SoftMax(Module):           # Output activation
    def forward(self, Z):
        return (np.exp(Z)/np.sum(np.exp(Z), axis = 0))        # Your code

    def backward(self, dLdZ):    # Assume that dLdZ is passed in
        return dLdZ

    def class_fun(self, Ypred):
        # Returns the index of the most likely class for each point as vector of shape (b,)
        return np.argmax(Ypred, axis=0)            # Your code

```

100.00%

You have infinitely many submissions remaining.

Here is the solution we wrote:

```
class SoftMax(Module):           # Output activation
    def forward(self, Z):
        # returns an array of indices of size (b,)
        return np.exp(Z) / np.sum(np.exp(Z), axis=0)

    def backward(self, dLdZ):    # Assume that dLdZ is passed in
        return dLdZ

    def class_fun(self, Ypred):  # Return class indices
        return np.argmax(Ypred, axis=0)
```

5.3) Loss Module - NLLM

We will now implement the NLL multiclass loss function assuming that the output activation function is softmax.

It will be useful to take advantage of the great simplification that happens when we have a softmax activation combined with NLLM as a loss function. In that case, that is, when our guess $g = \text{SM}(z)$ and desired output is y, then

$$\frac{\partial \mathcal{L}_{\text{NLLM}}(\text{SM}(z), y)}{\partial z} = (g - y)$$

You might find `np.log` to be helpful. You can find documentation online.

Note that `Ypred` in the code refers to the predicted y value that comes out of the neural network. This is the argument g to the loss function, and is also referred to as a^L (the activation of the last layer). For NLLM, it is a vector representing a probability distribution over the class of the input.

Hint: recall that the loss definition and that `Ypred` and `Y` may be associated with multiple data points. In the last step then, what type of multiplication would we want?

```
class NLL(Module):       # Loss
    def forward(self, Ypred, Y):
        # returns loss as a float
        self.Ypred = Ypred
        self.Y = Y
        return -np.sum(Y*np.log(Ypred))      # Your code

    def backward(self):  # Use stored self.Ypred, self.Y
        # note, this is the derivative of loss with respect to the input of softmax
        return self.Ypred - self.Y     # Your code

```

Run Code Submit View Answer **100.00%**

You have infinitely many submissions remaining.

5.4) Sequential

Now we will implement SGD following these specs:

```
*Randomly pick a data point Xt, Yt by using  `np.random.randint` to choose a random index into the data.
* Compute the predicted output Ypred for Xt with the forward method.
*Compute the loss for Ypred relative to Yt.
* Use the backward method to compute the gradients. (Hint, where should backprop start? Which module has a backward
that takes no input?)
*Use the sgd_step method to change the  weights.
* Repeat.
```

For context, we can construct a network and train it as follows:

```
# build a 3-layer network
net = Sequential([Linear(2,3), Tanh(),
                  Linear(3,3), Tanh(),
                  Linear(3,2), SoftMax()], NLL())
# train the network on data and labels
net.sgd(X, Y)
```

Add your code in in the place '# YOUR CODE HERE'. Do not edit other methods. You do not need to implement the loss function.

Hints: Recall the difference between indexing and slicing. Also make sure to use `self.loss` in your implementation of `sgd`.

A note on debugging: We have provided you with a colab that has a copy of the template below and a detailed set of outputs to check your implementation. **Trying to debug directly in the box below will not be a good experience.**

```python
class Sequential:
    def __init__(self, modules, loss):            # List of modules, loss module
        self.modules = modules
        self.loss = loss

    def sgd(self, X, Y, iters=100, lrate=0.005):  # Train
        D, N = X.shape
        for it in range(iters):
            i = np.random.randint(X.shape[1])
            x, y = X[:,i:i+1], Y[:, i:i+1]
            Ypred = self.forward(x)
            total_loss = self.loss.forward(Ypred, y)
            dz = self.loss.backward()
            self.backward(dz)
            self.sgd_step(lrate)

    # YOUR CODE HERE

    def forward(self, Xt):                        # Compute Ypred
        for m in self.modules: Xt = m.forward(Xt)
        return Xt

    def backward(self, delta):                    # Update dLdW and dLdW0
        # Note reversed list of modules
        for m in self.modules[::-1]: delta = m.backward(delta)

    def sgd_step(self, lrate):                    # Gradient descent step
        for m in self.modules: m.sgd_step(lrate)

    def print_accuracy(self, it, X, Y, cur_loss, every=250):
        # Utility method to print accuracy on full dataset, should
        # improve over time when doing SGD. Also prints COURSE/questions/nn loss,
        # which should decrease over time. Call this on each iteration
        # of SGD!
        if it % every == 1:
            cf = self.modules[-1].class_fun
            acc = np.mean(cf(self.forward(X)) == cf(Y))
            print('Iteration =', it, '  Acc =', acc, '  Loss =', cur_loss)

```

Run Code | Submit | View Answer **100.00%**

You have infinitely many submissions remaining.

6) Neural Network Packages

All subparts below are optional but encouraged

In the following problems, we will get experience with using a well-known neural network package for classification. We will be using PyTorch to build neural network models. Please use the PyTorch colab notebook, which comes with pre-installed PyTorch.

Background: PyTorch

We would like you to get familiar with PyTorch. It's a very popular ML framework for research and has similar fundamentals to other frameworks you might encounter in industry. We'll be mainly using `torch.nn` a PyTorch module designed to create and train neural networks, and `torch.optim` a PyTorch package that allows us to use out-of-the-box optimization algorithms. You should be able to specify a neural network architecture, e.g. what `nn.Linear(in_features=2, out_features=3, bias=True), nn.Softmax(-1)]` means. Read some details in the following pages to understand the components of the neural networks:

- *Containers and core layers*, read about `Linear`, `Sequential`, `Softmax`, and `ReLU` layers.

- *Optimizers*, read about `Adam`; we'll use `Adam` throughout this assignment, this is an adaptive step-size method that is a bit more sophisticated than having a fixed step size.
- *Losses*, read about `CrossEntropyLoss`, this combines what we call NLL loss with a log softmax layer in one single class.

You can complete the following problems with the information already provided; however, if you would like to dive further into PyTorch, you can read the documentation or review Deep learning with PyTorch: A 60 minute blitz.

Note that PyTorch assumes that each data point is a ROW vector and that a data set X with N points with D features is an NxD matrix. The code we have provided handles this, but you should be aware of this if you want to use the code in the future.

We will be working with a set of relatively simple datasets in 2D. These are all classification problems.

In this assignment we will focus on exploring the choice of network *architecture*. In particular, we will explore two classes of network architectures:

1. No hidden units - this is a linear classifier.
2. Fully connected (FC) layer(s) (`Linear` layers) - the standard approach to problems involving heterogeneous data.

If you look through the PyTorch documentation you'll see that there are an enormous number of choices that can be made to define a model. We're only going to look at a few, in particular:

- number and type of layers
- number of units per layer
- number of passes over the data (epochs)

Here are a few choices that we are **not** going to explore; we'll keep these choices fixed throughout:

- We'll use `ReLU` activation function for hidden units.
- We'll use `CrossEntropyLoss` as the loss which is a combination of the `softmax` activation and `NLLloss` on the output.
- We'll use `Adam` as the optimizer. This is a version of SGD that adapts the step size (learning rate) somewhat like Adadelta but a bit more sophisticated. We will use Adam's default parameters.

We have provided you a couple of basic functions to experiment with; you mostly have to define layers. We **strongly** recommend that you do your experiments using the PyTorch colab notebook file to keep track of your results. Alternatively, write scripts that can be run to produce your results and save them to a file. Have your script write out the settings for the experiment so that you can remember what they are when you go back to look at them.

2D Data

For the 2D datasets, we have provided the following function, which will train the specified neural network on the given dataset:

```
run_pytorch_2d(data_name, layers, epochs, split=0.25, display=False, verbose=False, trials=5)
```

where:

- `data_name` is a string, such as `'1'`, `'2'`, etc.
- `layers` is a network with one hidden layer with two hidden units, e.g. `[nn.Linear(in_features=2, out_features=2, bias=True), nn.ReLU(), nn.Linear(in_features=2, out_features=2, bias=True)]`
- `epochs` is an integer indicating how many times to go through the data in training
- `split` is a fraction of the training data to use for validation if a validation set is not defined
- `display` whether to display result plots
- `verbose` whether to print loss and accuracy (percent correctly labeled) each epoch
- `trials` is an integer indicating how many times to perform the training and testing

6.1)

Using the two-class 2D dataset `'2'`, run a network containing one hidden layer with two hidden units 20 times while setting `trials=1` and `epochs=200`. What are the minimum and maximum accuracies on the validation sets achieved by the network?

6.1.1)

Enter a decimal value indicating the minimum accuracy achieved by the network:

Check Formatting | Submit | View Answer | Ask for Help

You have infinitely many submissions remaining.

6.1.2)

Enter a decimal value indicating the maximum accuracy achieved by the network:

Check Formatting | Submit | View Answer | Ask for Help

You have infinitely many submissions remaining.

6.2)

Why do the accuracies in question 6.1 significantly vary?

- ○ Training did not run for sufficiently many epochs
- ○ With this architecture and dataset, gradient descent frequently gets stuck in poor local optima
- ○ Too few trials
- ○ This architecture is incapable of separating the dataset

Submit | View Answer | Ask for Help

You have infinitely many submissions remaining.

6.3)

Still using the two-class 2D dataset `'2'`, run a network containing one hidden layer with 5, 10, 20, and 100 hidden units while setting `trials=5` and `epochs=200`. What are the average accuracies on the validation sets achieved by the increasing hidden unit sizes?

6.3.1)

Enter a decimal value indicating the average accuracy achieved with 5 hidden units (please provide 4 significant digits):

Check Formatting | Submit | View Answer | Ask for Help

You have infinitely many submissions remaining.

6.3.2)

Enter a decimal value indicating the average accuracy achieved with 10 hidden units (please provide 4 significant digits):

Check Formatting | Submit | View Answer | Ask for Help

You have infinitely many submissions remaining.

6.3.3)

Enter a decimal value indicating the average accuracy achieved with 20 hidden units (please provide 4 significant digits):

Check Formatting | Submit | View Answer | Ask for Help

You have infinitely many submissions remaining.

6.3.4)

Enter a decimal value indicating the average accuracy achieved with 100 hidden units (please provide 4 significant digits):

Check Formatting | Submit | View Answer | Ask for Help

You have infinitely many submissions remaining.

6.4)

Still using the two-class 2D dataset `'2'`, run a simple network with no hidden layers while setting `trials=5` and `epochs=200`. What is the average accuracy on the validation sets achieved by the linear module?

6.4.1)

Enter a decimal value indicating the average accuracy achieved by the simple network (please provide 4 significant digits):

Check Formatting | Submit | View Answer | Ask for Help

You have infinitely many submissions remaining.

6.4.2)

Why is this accuracy lower than the accuracies from question 6.3?

○ This architecture needs more epochs to reach good performance

○ The architectures with hidden units are overfitting to the data

○ Without any hidden units, gradient descent is getting stuck in local optima

○ The dataset is not linearly separable

Submit | View Answer | Ask for Help

You have infinitely many submissions remaining.

6.5)

Still using the two-class 2D dataset `'2'`, implement a network with two hidden layers with 100 units each while setting `trials=5` and `epochs=300`. Which epoch did the validation loss reach minimum?

Enter an integer value indicating the epoch which achieved the minimum validation loss:

Check Formatting | Submit | View Answer | Ask for Help

You have infinitely many submissions remaining.

6.6)

Using the three-class 2D dataset `'3class'`, implement the three following networks: 1) no hiddens layers, 2) one hidden layer with 100 units followed by ReLU activation functions, and 3) two hidden layers with 100 units then 10 units each followed by ReLU activation functions. You should set `trials=1`, `split=0.5`, and `epochs=500`. What are the validation accuracies when each of these neural networks converges?

Hint: what would `output_size` have to be if we output a prediction between 3 classes?

6.6.1)

Enter a decimal value indicating the validation accuracy achieved implementing no hidden layers:

Check Formatting Submit View Answer Ask for Help

You have infinitely many submissions remaining.

6.6.2)

Enter a decimal value indicating the validation accuracy achieved implementing one hidden layer with 100 units:

Check Formatting Submit View Answer Ask for Help

You have infinitely many submissions remaining.

6.6.3)

Enter a decimal value indicating the validation accuracy achieved implementing two hidden layers with 100 units then 10 units:

Check Formatting Submit View Answer Ask for Help

You have infinitely many submissions remaining.

6.7)

Using your network from the previous question for implementing hidden layer (100, 10) while setting `trials=1`, `split=0.5`, and `epochs=300`, find the classification predictions for the following points:

```
[[-1,0], [1,0], [0,-11], [0,1], [-1,-1], [-1,1], [1,1], [1,-1]]
```

You will need to seed your code random number generator to get comparable results to ours. Please find the variable `deterministic` and set it to `True` (Completely reproducible results are not guaranteed across PyTorch releases, individual commits or different platforms, hence, we recommend that you use the colab we have provided you).

Hint: Read this Building Models with PyTorch section which gives an example of getting output `y` with neural network `lin` and points `x`. It will be helpful to look at torch.argmax for acquiring predictions.

Use your learned neural network to make predictions for these data points:

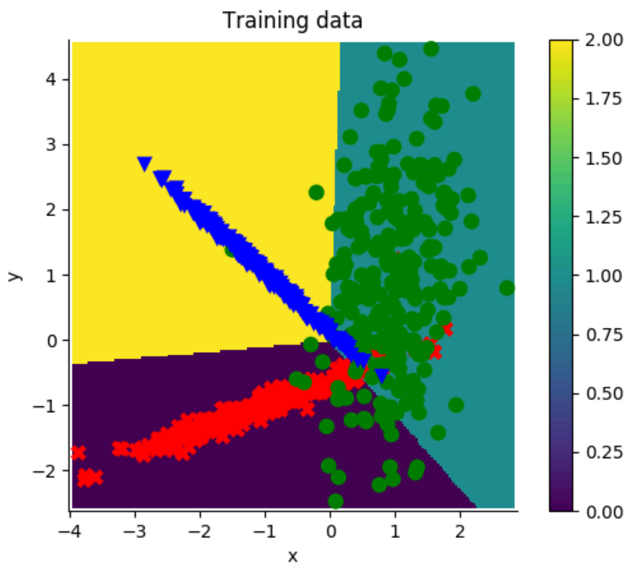
Enter a list of 8 class indices (0, 1, 2):

Check Formatting Submit View Answer Ask for Help

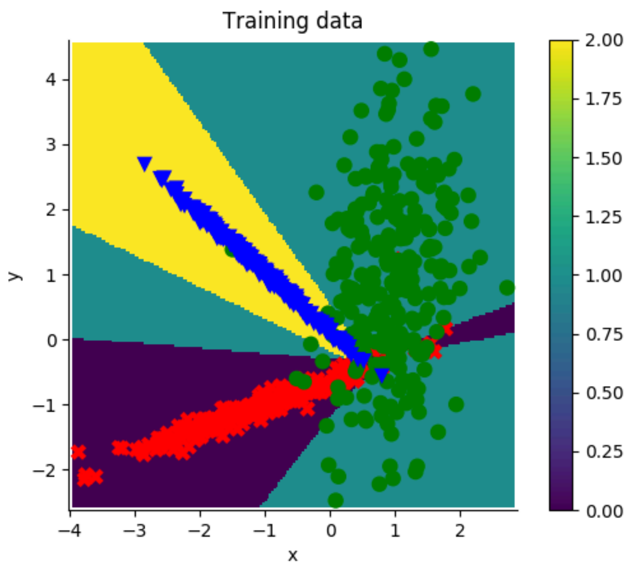
You have infinitely many submissions remaining.

6.8)

The decision boundary for the 3class data, using no hidden units (architecture 0) looks like this:



Here's another decision boundary for the 3class data obtained with lots of hidden units (architecture 4):



Mark all that are true:

- ☐ Increasing the complexity of the network enables fitting more complex training data.
- ☐ Increasing the number of training epochs enables fitting more complex training data.
- ☐ Increasing the complexity of the network and increasing the number of training epochs typically leads to plateauing or decreasing validation performance.

Submit | View Answer | Ask for Help

You have infinitely many submissions remaining.

Survey

(The form below is to help us improve/calibrate for future assignments; submission is encouraged but not required. Thanks!)

How did you feel about the **length** of this homework?

[✓ Too long.]

About right.

Too short.

How did you feel about the **difficulty** of this homework?

[✓ Too hard. We should tone it down.]

About right.

Too easy. I want more challenge.

Do you have any feedback or comments about any questions in this homework? Anything else you want us to know?

Submit