Assignment 7: Pytorch, Vanishing Gradients, and Convolutional Neural Networks

Machine Learning

Fall 2019

Learning Objectives

- Learn the basics of Pytorch
- Understand the vanishing gradient problem and how to fix it
- Learn about convolutional neural networks from a conceptual perspective
- Apply convolutional neural networks to a real dataset.

Prior Knowledge Utilized

- Multilayer perceptron
- Backpropagation

1 *Pytorch and Autograd Algorithms*

In this assignment you will be learning about another new Python library: pytorch. Pytorch is an open-source library designed for high-performance (meaning fast, rather than necessarily high accuracy) machine learning. There are a number of other similar frameworks out there. We chose pytorch due to its straightforward data handling, flexibility, and support for debugging. Below, we'll list some other frameworks and why we ultimately decided to go with pytorch.

- Keras also lets you construct high-performance neural networks in Python. The biggest downside in our eyes is that it hides away a lot of the details of how data moves through the network. This makes it a bit too abstract for our purposes (e.g., harder to debug and less flexible).
- TensorFlow is another package that is similar to pytorch. Tensorflow is more popular than pytorch, but it is a bit more complicated and has a steeper learning curve.
- sklearn (which you've gotten to use) has support for Multilayer Perceptrons (as we saw in the last notebook). That said, it doesn't support that many different types of networks, is hard to customize, and doesn't support high performance computation using GPUs (graphics processing units).
- numpy could also be an option. It is very flexible, but it is too low-level. Like sklearn it doesn't support high performance computation with GPUs.

Before you go through some resources on how to use pytorch, let's discuss what we are hoping to get from pytorch.

- The ability to automatically compute gradients of our loss function with respect to parameters of a neural network!
- The ability to train neural networks on a GPU (for performance boosts of up to 20 times). Luckily, Google is nice enough to let you use their GPUs through Colab!

- Tools for debugging common problems with neural networks.
- A module called torchvision for working with images in neural networks.

Go through the following tutorials for pytorch. This will give you the barebones. Later in this document you'll explore pytorch again via a companion notebook.

External Resource(s) (15 minutes)

We recommend you go through these with Colab open so you can try the blocks of code yourself.

- Tensor Tutorial (on NB)
- Autograd Tutorial (on NB)

External Resource(s) (TBD minutes)

Now that you have the barebones basics for Pytorch, let's look back at our old friend the logistic regression model. In this companion notebook you'll play with an implementation of logistic regression using pytorch and compare it with sklearn's implementation of logistic regression. In the notebook we'll use the Titanic data as our example.

2 Backpropagation and the Problem of Vanishing (or Exploding) Gradients

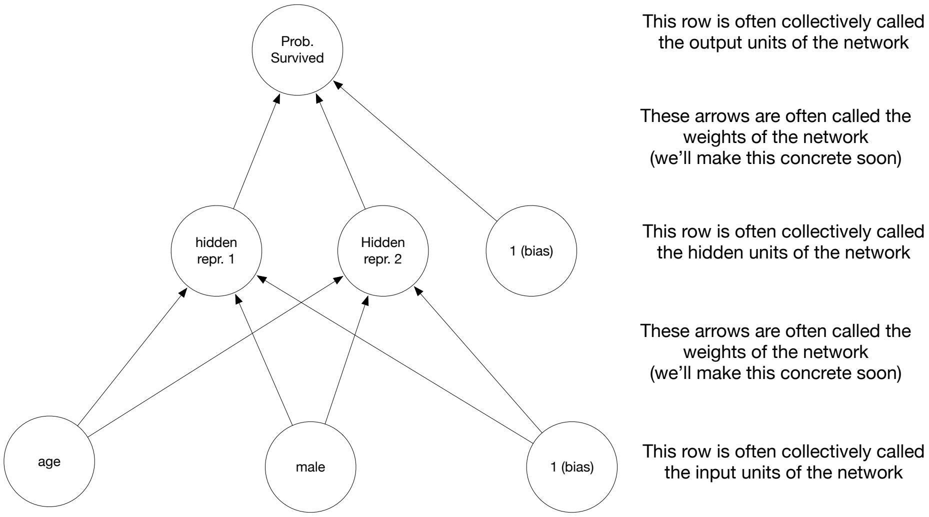
One possible reaction to reading about the autograd capabilities in pytorch is to think "Why didn't Paul and Sam tell me about this before? Why have we been wasting our time doing all this math to compute gradients?" There are two really good (in our opinion) answers to this question.

1. Knowing how backpropagation works gives you a more powerful mental model of how these networks are trained, how to diagnose failures in training, and potentially how to fix these failures.
2. The math you learned along the way is useful even outside of machine learning.

One particularly important way in which knowing backpropagation helps you understand how to train a better network is when confronting something known as the *vanishing gradient problem* (or its more dramatic sounding counterpart the *exploding gradient problem*). You'll be reading all about it in just a bit, but before we dive in let's review the multi-layer perceptron (MLP). If you feel good about the MLP, skip the next box.

Recall: Multi-layer perceptron

The multi-layer perceptron consists of a bunch of single-layer perceptrons stacked on top of each other. Last assignment we saw a particular special case of the MLP where we take logistic regression models and stack them. A cartoon version of the network applied to the titanic dataset is shown below.



This row is often collectively called the output units of the network

These arrows are often called the weights of the network (we'll make this concrete soon)

This row is often collectively called the hidden units of the network

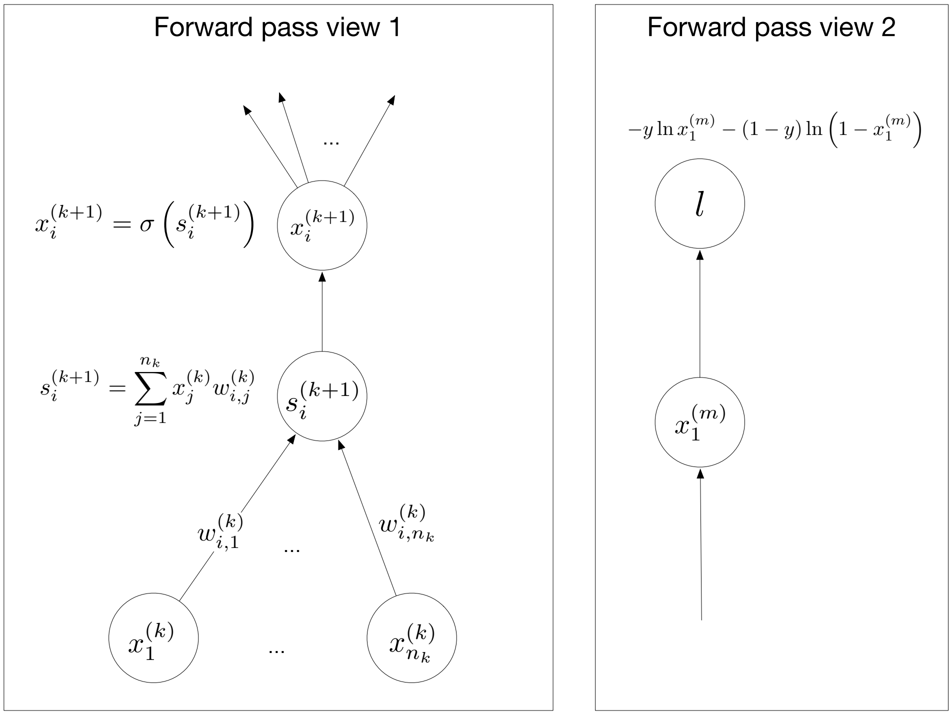
These arrows are often called the weights of the network (we'll make this concrete soon)

This row is often collectively called the input units of the network

We saw in the companion notebook that such networks are capable of learning useful internal representations for prediction. To make this figure more precise, we can zoom in on how data propagates from lower layers to higher layers in the network (view 1 in the figure below) and ultimately to the loss function (view 2 in the figure below).

Before presenting the figure, for your convenience here is a notation guide for the figure.

l	the loss function for the network (log loss in this case)
m	the number of layers in the network
n_k	the number of nodes in the kth layer
$x_j^{(k)}$	The jth node in the kth layer of the network ($k = 1$ is the input and $k = m$ is the output)
$\mathbf{x^{(k)}}$	The nodes at the kth layer in vector form
$s_i^{(k)}$	the ith summation node in the kth layer
$w_{i,j}^{(k)}$	the weight connecting the jth node in layer k to the ith node in layer $k+1$
$\mathbf{w_i^{(k)}}$	the vector of weights to the ith summation node in layer $k+1$.


Forward pass view 1
$x_i^{(k+1)} = \sigma\left(s_i^{(k+1)}\right)$
$x_i^{(k+1)}$
...
$s_i^{(k+1)} = \sum_{j=1}^{n_k} x_j^{(k)} w_{i,j}^{(k)}$
$s_i^{(k+1)}$
$w_{i,1}^{(k)}$
...
$w_{i,n_k}^{(k)}$
$x_1^{(k)}$
...
$x_{n_k}^{(k)}$
Forward pass view 2
$-y \ln x_1^{(m)} - (1-y)\ln\left(1 - x_1^{(m)}\right)$
l
$x_1^{(m)}$

In the previous assignment you derived the backpropagation algorithm, which can be used to compute the gradient of the weights in an MLP with respect to the loss function. What's beautiful about this formula is that it works for networks of arbitrary depth. You may have heard of deep learning, which is where networks of dozens or even hundreds of layers are trained on a particular task. Despite their complexity, these networks are typically trained using the backpropagation algorithm you met in the last assignment!

Despite the relative simplicity of the backpropagation equations, there are some dangers lurking when applying the algorithm to deep networks (m very large). The most common issues that one encounters are **vanishing and exploding gradients**. A vanishing gradient is when the gradient of the loss with respect to the weights in a layer becomes vanishingly small as you move from the output layer towards the input layer. An exploding gradient is the opposite problem (when the gradient of the loss with respect to the weights in a layer becomes exceedingly large as you move from the output layer towards the input layer).

Exercise 1 (45 minutes) or (105 minutes including optional part)

Before starting this exercise, go and learn about the vanishing gradient problem. Here are a few resources to look at. You do not need to read all of them, so please pick the ones that seem the most inline with how you learn.

External Resource(s)

- Rohan Kapur's Intuitive Explanation of the Vanishing Gradient Problem (on NB)
- Video resource: Deep Learning Simplified: An Old Problem

Last assignment we derived the following backpropagation equations.

$$(\nabla_{\mathbf{w}_{\mathbf{i}}^{(\mathbf{k})}})l = \mathbf{x}^{(\mathbf{k})}\sigma\left(s_i^{(k+1)}\right)\left(1-\sigma\left(s_i^{(k+1)}\right)\right)\frac{\partial l}{\partial x_i^{(k+1)}} \qquad \text{gives us the gradient of the weights going into a unit} \tag{1}$$

$$\frac{\partial l}{\partial x_i^{(k)}} = \sum_{j=1}^{n_{k+1}} w_{j,i}^{(k)}\sigma\left(s_j^{k+1}\right)\left(1-\sigma\left(s_j^{k+1}\right)\right)\frac{\partial l}{\partial x_j^{(k+1)}} \qquad \text{recursively gives the partial of the loss w.r.t. the units} \tag{2}$$

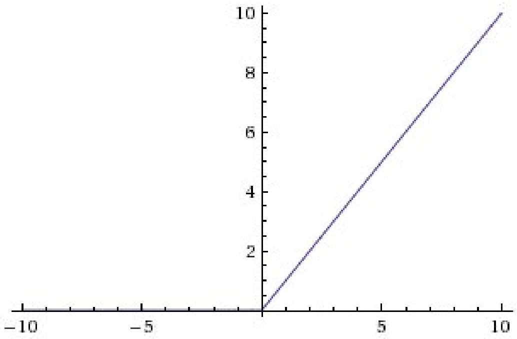
$$\frac{\partial l}{\partial x_1^{(m)}} = -y\frac{1}{x_1^{(m)}} + (1-y)\frac{1}{1-x_1^{(m)}} \qquad \text{provides the base case of the recursion} \tag{3}$$

(a) If we examine these equations, we can understand where the vanishing gradient problem comes from. In particular, Equation 2 provides the crucial recursive formula for defining the partial derivatives of the loss with respect to nodes in the network in terms of the same partial derivatives for nodes deeper in the network (i.e., closer to the output). Based on Equation 2, what aspects of the network might contribute to or counteract the vanishing gradient problem? Consider things such as the values of the summation units $s_j^{(k+1)}$, the values of the weights, and the number of units at each layer of the network (the n_k's).

(b) It turns out that the sigmoid function is only one viable choice for transforming the summation units into the values of the units for the next layer. In fact, many different non-linear functions can be used (e.g., tanh). The function used in this capacity is called an *activation function*. If we let a refer to our activation function, then Equation 2 becomes

$$\frac{\partial l}{\partial x_i^{(k)}} = \sum_{j=1}^{n_{k+1}} w_{j,i}^{(k)} a'\left(s_j^{k+1}\right)\frac{\partial l}{\partial x_j^{(k+1)}} \quad . \tag{4}$$

One very popular choice of activation function is called rectified linear (or ReLu for *rectified linear unit*). A graph of the ReLu activation function is shown below.



How might choosing the ReLu as an activation function help with the vanishing gradient problem in comparison to choosing the sigmoid as the activation function (use the backpropagation equations to arrive at your answer)?

(c) **Going Beyond (optional)** Using pytorch, perform a computational investigation of the vanishing gradient problem. We suggest the following steps (note: we have our own version of this in the solutions if you want to check it out):

- Create an MLP where the number of layers is passed in as a parameter.
- Before doing any training of the network, feed an arbitrary input into the network, compute the loss, and then compute the gradient of the loss with respect to the weights at each level of the network.
- Summarize the partial derivatives of the weights at each level in the network (e.g., take the mean of the absolute values) and plot this quantity as a function of k (the layer number). It may be helpful to use a log scale for the y-axis.

3 Convolutions and Image Filtering

Depending on your math background you may have seen the convolution operation before. There are various types of convolutions, including convolutions with continuous functions, discrete functions, in 1D, in 2D, or in higher dimensions. In this class we're only going to be talking about discrete, 2D convolutions. It turns out that such convolutions provide a compelling way to think about processing data with intrinsically 2D structure (such as images).

Check out at least one of the resources below to learn how convolutions can be used to filter and transform images.

External Resource(s) (30 minutes)

- Image Filtering (on NB)
- Video resource: How blurs and filters work

Exercise 2 (10 minutes)

This is not an exercise per se. On the quiz you should mark whether or not you were able to engage with the readings and get a basic grasp of the topic. As a basic guide, you should have some familiarity with the following ideas.

- What a 2D convolutional kernel is and how it is applied to a 2D matrix (e.g., an image).
- What happens when applying the kernel around the edges of the 2D matrix?
- Understand how to reason about what a kernel does to an image based on the form of the kernel.

4 Convolutional Neural Networks

We now come to a particular type of neural network called a convolutional neural network. These networks supplement the perceptron layers we've seen thus far with convolutional layers that apply various filtering operations to the data. These networks are very well-suited for working with images since they are able to customize their filtering operations to the dataset in a way that maximizes their ability to solve the task. There are lots of great resources on convolutional neural networks out there. Please don't feel like you have to go through all of these.

External Resource(s) (60 minutes)

- Intuitive Explanation of Convnets (on NB) (this one is a very well done high-level overview)
- Beautiful interactive visualization of a Convnet trained to recognize handwritten digits (this is great, especially

for visual learners. Make sure to click on the activations at a layer to see the learned filter. Don't miss this one!)

- Convnet.js (convnets training in your browser?!? Another good one for visual learners)
- Convolutional Neural Networks Explained (on NB) (this one is quite similar to the "intuitive explanation of convnets." It's a little shorter though.)
- Lecture 5 from CS231 at Stanford (if you want a lecture, this one is good. it starts getting a bit too low-level around the 35 minute mark. there are some nice slides on neural network history at the beginning).

Exercise 3 (15 minutes)

Open up Convnet.js and try it on either MNIST or CIFAR-10. Watch the network train. Interpret as much of what you see as possible. If you don't understand something, ask about it on NB (Convnet.js on MNIST or Convnet.js on CIFAR-10)

Open in Colab

Pytorch and Titanic: Connecting the Math to Coding

Now that you've seen the very basics of how to use `pytorch`, we're going to see how to apply it to a machine learning problem. Along the way we'll make connections back to the Titanic dataset and help solidify your understanding of the connection between the math we've been learning and the code we'll be writing using `pytorch`.

To get started, let's load our trusty Titanic dataset.

In [1]:

```
import gdown
import numpy as np
import pandas as pd

gdown.download('https://drive.google.com/uc?authuser=0&id=1XIFiL3WxxR6M2nWgADi3xWvuRO6A-Ov8&export=download', 'titanic_train.csv', False)
df = pd.read_csv('titanic_train.csv')
df
```

```
Downloading...
From: https://drive.google.com/uc?authuser=0&id=1XIFiL3WxxR6M2nWgADi3xWvuRO6A-Ov8&export=download
To: /Users/pruvolo/Documents/assignments/Module 1/07/titanic_train.csv
100%|██████████| 61.2k/61.2k [00:00<00:00, 1.95MB/s]
```

Out[1]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
5	6	0	3	Moran, Mr. James	male	NaN	0	0	330877	8.4583	NaN	Q
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.8625	E46	S
7	8	0	3	Palsson, Master. Gosta Leonard	male	2.0	3	1	349909	21.0750	NaN	S
8	9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	27.0	0	2	347742	11.1333	NaN	S
9	10	1	2	Nasser, Mrs. Nicholas (Adele Achem)	female	14.0	1	0	237736	30.0708	NaN	C
10	11	1	3	Sandstrom, Miss. Marguerite Rut	female	4.0	1	1	PP 9549	16.7000	G6	S
11	12	1	1	Bonnell, Miss. Elizabeth	female	58.0	0	0	113783	26.5500	C103	S
12	13	0	3	Saundercock, Mr. William Henry	male	20.0	0	0	A/5. 2151	8.0500	NaN	S
13	14	0	3	Andersson, Mr. Anders Johan	male	39.0	1	5	347082	31.2750	NaN	S
14	15	0	3	Vestrom, Miss. Hulda Amanda Adolfina	female	14.0	0	0	350406	7.8542	NaN	S
15	16	1	2	Hewlett, Mrs. (Mary D Kingcome)	female	55.0	0	0	248706	16.0000	NaN	S
16	17	0	3	Rice, Master. Eugene	male	2.0	4	1	382652	29.1250	NaN	Q
17	18	1	2	Williams, Mr. Charles Eugene	male	NaN	0	0	244373	13.0000	NaN	S
18	19	0	3	Vander Planke, Mrs. Julius	female	31.0	1	0	345763	18.0000	NaN	S

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
				(Emelia Maria Vande...								
19	20	1	3	Masselmani, Mrs. Fatima	female	NaN	0	0	2649	7.2250	NaN	C
20	21	0	2	Fynney, Mr. Joseph J	male	35.0	0	0	239865	26.0000	NaN	S
21	22	1	2	Beesley, Mr. Lawrence	male	34.0	0	0	248698	13.0000	D56	S
22	23	1	3	McGowan, Miss. Anna "Annie"	female	15.0	0	0	330923	8.0292	NaN	Q
23	24	1	1	Sloper, Mr. William Thompson	male	28.0	0	0	113788	35.5000	A6	S
24	25	0	3	Palsson, Miss. Torborg Danira	female	8.0	3	1	349909	21.0750	NaN	S
25	26	1	3	Asplund, Mrs. Carl Oscar (Selma Augusta Emilia...	female	38.0	1	5	347077	31.3875	NaN	S
26	27	0	3	Emir, Mr. Farred Chehab	male	NaN	0	0	2631	7.2250	NaN	C
27	28	0	1	Fortune, Mr. Charles Alexander	male	19.0	3	2	19950	263.0000	C23 C25 C27	S
28	29	1	3	O'Dwyer, Miss. Ellen "Nellie"	female	NaN	0	0	330959	7.8792	NaN	Q
29	30	0	3	Todoroff, Mr. Lalio	male	NaN	0	0	349216	7.8958	NaN	S
...	...	...	...	...	...	...	...	...	...	...	...	...
861	862	0	2	Giles, Mr. Frederick Edward	male	21.0	1	0	28134	11.5000	NaN	S
862	863	1	1	Swift, Mrs. Frederick Joel (Margaret Welles Ba...	female	48.0	0	0	17466	25.9292	D17	S
863	864	0	3	Sage, Miss. Dorothy Edith "Dolly"	female	NaN	8	2	CA. 2343	69.5500	NaN	S
864	865	0	2	Gill, Mr. John William	male	24.0	0	0	233866	13.0000	NaN	S
865	866	1	2	Bystrom, Mrs. (Karolina)	female	42.0	0	0	236852	13.0000	NaN	S
866	867	1	2	Duran y More, Miss. Asuncion	female	27.0	1	0	SC/PARIS 2149	13.8583	NaN	C
867	868	0	1	Roebling, Mr. Washington Augustus II	male	31.0	0	0	PC 17590	50.4958	A24	S
868	869	0	3	van Melkebeke, Mr. Philemon	male	NaN	0	0	345777	9.5000	NaN	S
869	870	1	3	Johnson, Master. Harold Theodor	male	4.0	1	1	347742	11.1333	NaN	S
870	871	0	3	Balkic, Mr. Cerin	male	26.0	0	0	349248	7.8958	NaN	S
871	872	1	1	Beckwith, Mrs. Richard Leonard (Sallie Monypeny)	female	47.0	1	1	11751	52.5542	D35	S
872	873	0	1	Carlsson, Mr. Frans Olof	male	33.0	0	0	695	5.0000	B51 B53 B55	S
873	874	0	3	Vander Cruyssen, Mr. Victor	male	47.0	0	0	345765	9.0000	NaN	S
874	875	1	2	Abelson, Mrs. Samuel (Hannah Wizosky)	female	28.0	1	0	P/PP 3381	24.0000	NaN	C
875	876	1	3	Najib, Miss. Adele Kiamie "Jane"	female	15.0	0	0	2667	7.2250	NaN	C
876	877	0	3	Gustafsson, Mr. Alfred Ossian	male	20.0	0	0	7534	9.8458	NaN	S
877	878	0	3	Petroff, Mr. Nedelio	male	19.0	0	0	349212	7.8958	NaN	S
878	879	0	3	Laleff, Mr. Kristo	male	NaN	0	0	349217	7.8958	NaN	S
879	880	1	1	Potter, Mrs. Thomas Jr (Lily Alexenia Wilson)	female	56.0	0	1	11767	83.1583	C50	C
880	881	1	2	Shelley, Mrs. William (Imanita Parrish Hall)	female	25.0	0	1	230433	26.0000	NaN	S
881	882	0	3	Markun, Mr. Johann	male	33.0	0	0	349257	7.8958	NaN	S
882	883	0	3	Dahlberg, Miss. Gerda Ulrika	female	22.0	0	0	7552	10.5167	NaN	S
883	884	0	2	Banfield, Mr. Frederick James	male	28.0	0	0	C.A./SOTON 34068	10.5000	NaN	S
884	885	0	3	Sutehall, Mr. Henry Jr	male	25.0	0	0	SOTON/OQ 392076	7.0500	NaN	S
885	886	0	3	Rice, Mrs. William (Margaret Norton)	female	39.0	0	5	382652	29.1250	NaN	Q

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
886	887	0	2	Montvila, Rev. Juozas	male	27.0	0	0	211536	13.0000	NaN	S
887	888	1	1	Graham, Miss. Margaret Edith	female	19.0	0	0	112053	30.0000	B42	S
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	W./C. 6607	23.4500	NaN	S
889	890	1	1	Behr, Mr. Karl Howell	male	26.0	0	0	111369	30.0000	C148	C
890	891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.7500	NaN	Q

891 rows × 12 columns

In the Assignment 6 companion notebook we fit two models:

1. We used the `Age` and `Sex` columns along with a synthetic feature called `is young male` as inputs to a logistic regression model in order to predict whether someone survived.
2. We used just the `Age` and `Sex` as inputs to a multilayer perceptron model in order to predict whether someone survived.

In this notebook we'll be implementing both of these models in `pytorch`. To get started, let's perform the following data processing / cleaning steps.

1. Get rid of any passengers where we don't know their age (don't do this in a real machine learning application as it will skew your results).
2. Convert the `Sex` column to a dummy variable called `male` that will take on value 1 if the passenger is male and 0 otherwise.
3. Create the `is young male` column that will be 1 for males under the age of 5 and 0 for everyone else.

In [2]:

```
# get rid of null values for age since this is just an illustrative example.
# this would not be a good thing to do if we were trying to evalutate the
# performance of a model.
df_filtered = df[['Age', 'Sex', 'Survived']].dropna()
is_young_male = ((df_filtered['Sex'] == 'male') & (df_filtered['Age'] < 5)).astype(int)
is_young_male.name = 'is_young_male'
experiment_1_data = pd.concat((pd.get_dummies(df_filtered['Sex'], drop_first=True),
df_filtered['Age'], is_young_male), axis=1)
experiment_1_outputs = df_filtered['Survived']
experiment_1_data
```

Out[2]:

	male	Age	is_young_male
0	1	22.0	0
1	0	38.0	0
2	0	26.0	0
3	0	35.0	0
4	1	35.0	0
6	1	54.0	0
7	1	2.0	1
8	0	27.0	0
9	0	14.0	0
10	0	4.0	0
11	0	58.0	0
12	1	20.0	0
13	1	39.0	0
14	0	14.0	0
15	0	55.0	0
16	1	2.0	1
18	0	31.0	0
20	1	35.0	0

	male	Age	is_young_male
21	1	34.0	0
22	0	15.0	0
23	1	28.0	0
24	0	8.0	0
25	0	38.0	0
27	1	19.0	0
30	1	40.0	0
33	1	66.0	0
34	1	28.0	0
35	1	42.0	0
37	1	21.0	0
38	0	18.0	0
...	...	...	...
856	0	45.0	0
857	1	51.0	0
858	0	24.0	0
860	1	41.0	0
861	1	21.0	0
862	0	48.0	0
864	1	24.0	0
865	0	42.0	0
866	0	27.0	0
867	1	31.0	0
869	1	4.0	1
870	1	26.0	0
871	0	47.0	0
872	1	33.0	0
873	1	47.0	0
874	0	28.0	0
875	0	15.0	0
876	1	20.0	0
877	1	19.0	0
879	0	56.0	0
880	0	25.0	0
881	1	33.0	0
882	0	22.0	0
883	1	28.0	0
884	1	25.0	0
885	0	39.0	0
886	1	27.0	0
887	0	19.0	0
889	1	26.0	0
890	1	32.0	0

714 rows × 3 columns

Next, we'll go ahead and build our logistic regression model just as we did in the assignment 6 companion notebook. The only small twist we will introduce is turning off the ridge term of the model so that it will make comparing the results from this analysis to what we do in pytorch easier.

In [3]:

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(solver='lbfgs', penalty='none') # setting solve silences annoying warning
model.fit(experiment_1_data, experiment_1_outputs)
print("coefs", model.coef_)
print("intercept", model.intercept_)
```

```
coefs [[-2.67342472  0.00808732  2.38307187]]
intercept [0.9018627]
```

Notebook Exercise 1 (10 minutes)

This is a bit of review. Provide an interpretation for the coefficients learned by the logistic regression model (e.g., what do they mean for the prediction of whether a passenger would survive).

Solution

TODO

Reimplementation in Pytorch

Next, we'll be using `pytorch` to implement our own version of logistic regression! When creating a neural network model (remember, we can think of logistic regression as a perceptron with 2 layers, an input and an output), you create a class that inherits from `nn.Module`. We'll give you an implementation of logistic regression and then give a detailed breakdown of the key lines.

In [4]:

```
from torch import nn
import torch
from torch.autograd import Variable

class LogisticRegressionPytorch(nn.Module):
    def __init__(self):
        super(LogisticRegressionPytorch, self).__init__()
        self.linear = nn.Linear(3,1)

    def forward(self, X):
        """ Propagate data through the network.

            This model first applies the linear layer and then a sigmoid
        """
        X = self.linear(X)
        return torch.sigmoid(X)
```

Here is a breakdown of some of the key lines in this implementation.

Inherit from the super class `nn.Module`:

```
class LogisticRegressionPytorch(nn.Module):
```

Since we're inheriting from `nn.Module`, we need to make sure to call the `__init__` method of `nn.Module` when initializating our class.

```
super(LogisticRegressionPytorch, self).__init__()
```

The linear layer will store the weight vector of our model. The 3 arises from the fact that our model will have 3 inputs (age, male, and is young male). It is very important that you store your layers as attributes of your class. This is how `pytorch` knows about them and can optimize them. If you need to have a list of layers, look into `nn.ModuleList`.

```
self.linear = nn.Linear(3,1)
```

The `forward` function is the heart of the model. It runs input data through the network and returns the output. Writing such functions usually amounts to passing data between the various layers that were created in the `__init__` method. The syntax for this is a little funny. For instance, in the code below, `self.linear(X)` implicitly calls the `forward` function of the `nn.Linear` class. Yes, we find this kind of weird, but that's how it's done in `pytorch`. Thius is really just a syntactic quirk rather than anything substantive that you need to worry about. The last step of the function involves applying the `sigmoid` and returning the result.

Next, we'll show how to pass some data into the model.

```
def forward(self, X):
    """ Propagate data through the network.

        This model first applies the linear layer and then a sigmoid
    """
    X = self.linear(X)
    return torch.sigmoid(X)
```

In [5]:

```
# sample_data represents a male passenger who is 10 years old
sample_data = Variable(torch.FloatTensor([1.0, 10.0, 0.0]))
model_pytorch = LogisticRegressionPytorch()
model_pytorch(sample_data)
```

Out[5]:

```
tensor([0.1327], grad_fn=<SigmoidBackward>)
```

The code we computed the probability that the specific passenger would survive. It is very important to realize that right now the model *has not been trained*. This means that we don't expect the output of the model to make any sense (although it might just by chance). If you rerun the code repeatedly, you'll get different results due to the fact that the weights are initialized randomly.

Next, we're going to actually train the network! This is where things get interesting. We'll have you run the code and then point out a couple of key components.

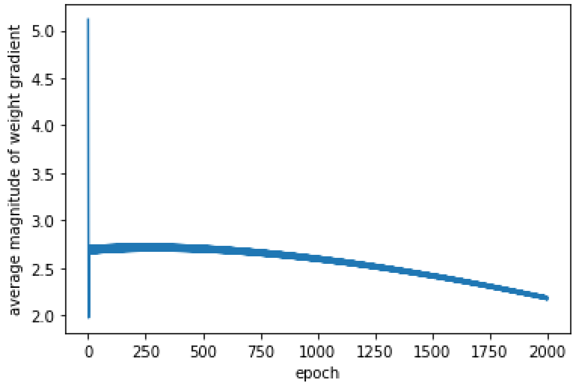
In [6]:

```
model_pytorch = LogisticRegressionPytorch()
model_pytorch.train()
optimizer = torch.optim.SGD(model_pytorch.parameters(), lr=0.01)
criterion = torch.nn.BCELoss()
grad_magnitudes = []

X_data = Variable(torch.Tensor(np.array(experiment_1_data)))
y_data = Variable(torch.Tensor(np.array(experiment_1_outputs)))
for epoch in range(2000):
    optimizer.zero_grad()
    # Forward pass
    y_pred = model_pytorch(X_data)
    # Compute Loss
    loss = criterion(y_pred, y_data)
    # Backward pass
    loss.backward()
    for name, param in model_pytorch.named_parameters():
        if name == 'linear.weight':
            grad_magnitudes.append(np.abs(param.grad.numpy()).mean())

    if epoch % 1000 == 0:
        print("epoch", epoch)
        for name, param in model_pytorch.named_parameters():
            print(name, "value", param.data, "gradient", param.grad)
    optimizer.step()

import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(grad_magnitudes)
plt.xlabel('epoch')
plt.ylabel('average magnitude of weight gradient')
plt.show()
```

```
epoch 0
linear.weight value tensor([[-0.0938,  0.0594, -0.3953]]) gradient tensor([[ 4.0462e-01,
1.4941e+01, -6.0478e-03]])
linear.bias value tensor([0.2207]) gradient tensor([0.4358])
```

```
/anaconda3/lib/python3.6/site-packages/torch/nn/functional.py:2016: UserWarning: Using a target si
ze (torch.Size([714])) that is different to the input size (torch.Size([714, 1])) is deprecated. P
ze (torch.Size([714])) that is different to the input size (torch.Size([714, 1])) is deprecated. P
lease ensure they have the same size.
  "Please ensure they have the same size.".format(target.size(), input.size()))
```

```
epoch 1000
linear.weight value tensor([[-1.0277,  0.0269, -0.2915]]) gradient tensor([[ 0.2052,  7.6471, -0.0
119]])
linear.bias value tensor([0.3365]) gradient tensor([0.2003])
```



Put the model into training mode (this only affects certain models that behave differently during training and testing). Even though it doesn't affect our logistic regression model, calling the `.train` function is a good habit to get into.

```
model_pytorch.train()
```

Create an optimizer that will tune the parameters of the network. Here we are using an algorithm called *stochastic gradient descent*. It's a very popular choice for an optimizer. Similarly to gradient descent it optimizes a function by stepping down the gradient (step size is given by learning rate or `lr`). Where it differs from normal gradient descent is that it doesn't necessarily use all of the data to compute the gradient. If you have a big dataset, it might only use a small number of data points, compute gradient over those, and then step down that estimate of the gradient. The collection of datapoints used for estimating the gradient is called a *mini batch*. In this example we are using the entire dataset each time, so we are really doing normal gradient descent.

```
optimizer = torch.optim.SGD(model_pytorch.parameters(), lr=0.01)
```

The criterion defines the loss function we are minimizing. When the training outputs are binary, the `BCELoss` function is equivalent to the log loss that we have been using in this course.

```
criterion = torch.nn.BCELoss()
```

In order to operate on data in pytorch, you have to convert any matrix or vector data into a pytorch variable. This should be familiar based on the tutorial you went through earlier.

```
X_data = Variable(torch.Tensor(np.array(experiment_1_data)))
y_data = Variable(torch.Tensor(np.array(experiment_1_outputs)))
```

An *epoch* in neural network training is a single pass through the data. In this case we are taking a single gradient step on the whole dataset, so the number of epochs is the same as the number of gradient steps.

```
for epoch in range(200):
```

This tells the optimizer to throw away any gradients it has accumulated from previous data (do not forget to call this!!!).

```
optimizer.zero_grad()
```

Apply the forward model to get predictions.

```
y_pred = model_pytorch(X_data)
```

Calculate the loss of the model by comparing its predictions with the actual outputs.

```
loss = criterion(y_pred, y_data)
```

Use backpropagation to compute the gradient of all of the model parameters with respect to the loss.

```
loss.backward()
```

Calculate the gradient magnitudes so we can make a plot after training.

```
for name, param in model_pytorch.named_parameters():
    if name == 'linear.weight':
        grad_magnitudes.append(np.abs(param.grad.numpy()).mean())
```

Print out the values of the parameters and the gradient of the parameters with respect to the loss every 50 epochs.

```
if epoch % 50 == 0:
    print("epoch", epoch)
    for name, param in model_pytorch.named_parameters():
        print(name, "value", param.data, "gradient", param.grad)
```

Perform the gradient step.

```
optimizer.step()
```

Notebook Exercise 2 (30 minutes)

(a) Explain the output you see when you run the previous code cell. How are the weights changing over time? How is the gradient changing over time? Is the algorithm close to converging (i.e., computing the optimal solution)? How would you know if it has converged?

(b) Increase the number of epochs until you get convergence. How many did it take?

(c) Tune the learning rate to some other values. How does this change the algorithm's behavior?

(d) Change the optimizer to the ADAM optimizer by swapping out the previous optimizer with this new line of code.

```
optimizer = torch.optim.Adam(model_pytorch.parameters())
```

Roughly many epochs does it take to reach convergence now?

Solution

TODO

Multilayer Perceptron

Now that we've shown you how to implement the logistic regression model, we want you to implement the MLP model from the previous companion notebook. Remember, the MLP had 2 input features (we didn't use `is young male` as an input) and 2 hidden units. We'll provide you with the skeleton of the code as well as some code to generate the visualization from the previous notebook.

In [7]:

```
# start from this and modify it
class LogisticRegressionPytorch(nn.Module):
    def __init__(self):
        super(LogisticRegressionPytorch, self).__init__()
        self.linear = nn.Linear(3,1)

    def forward(self, X):
        """ Propagate data through the network.

            This model first applies the linear layer and then a sigmoid
        """
        X = self.linear(X)
        return torch.sigmoid(X)
```

Notebook Exercise 3 (20 minutes + 20 minutes of optional work)

Non-optional: modify the code above regression class to create a class called TitanicMLP that has 2 input units and 2 hidden units. Train your network on the Titanic dataset. We have defined a function called `visualize_model_probs` for visualizing the probability plot tha we saw in the last companion notebook (this code should be run after the model is done training).

Optional: visualize the hidden unit representations in the network (similar to what we did in the companion notebook last time).

In [8]:

```
def visualize_model_probs(model):
    xx, yy = np.mgrid[-.1:1.1:.01, 0:85:.1]
    grid = np.c_[xx.ravel(), yy.ravel()]
    probs = model(Variable(torch.Tensor(grid))).detach().numpy().reshape(xx.shape)

    f, ax = plt.subplots(figsize=(8, 6))
    contour = ax.contourf(xx, yy, probs, 25, cmap="RdBu",
                          vmin=0, vmax=1)
    ax_c = f.colorbar(contour)
    ax_c.set_label("$P(survived)$")
    ax_c.set_ticks([0, .25, .5, .75, 1])

    ax.scatter(experiment_1_data['male'], experiment_1_data['Age'], c=experiment_1_outputs, s=50,
            cmap="RdBu", vmin=-.2, vmax=1.2,
            edgecolor="white", linewidth=1)

    ax.set(xlim=(-.1, 1.1),
        ylim=(0, 85),
        xlabel="is male", ylabel="age (years)")
    plt.show()
```

In [9]:

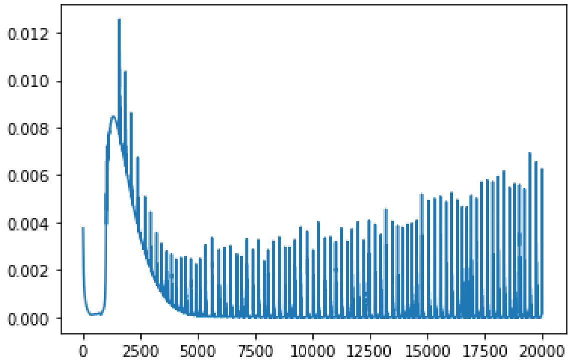
```
class TitanicMLP(nn.Module):
    def __init__(self):
        super(TitanicMLP, self).__init__()
        self.linear_1 = nn.Linear(2,2)
        self.linear_2 = nn.Linear(2,1)

    def forward(self, X):
        """ Propagate data through the network.

            This model first applies the linear layer, a sigmoid, a linear
            layer, and finally a sigmoid
        """
        X = self.linear_1(X)
        X = torch.sigmoid(X)
        X = self.linear_2(X)
        return torch.sigmoid(X)

    def hidden(self, X):
        """ Propagate data to the hidden layer """
        X = self.linear_1(X)
        return torch.sigmoid(X)
```

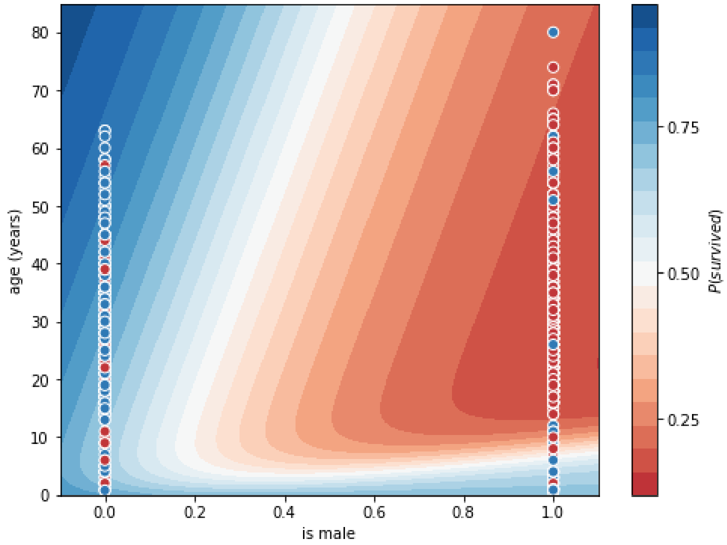
In [10]:

```
mlp_pytorch = TitanicMLP()
mlp_pytorch.train()
optimizer = torch.optim.Adam(mlp_pytorch.parameters())
criterion = torch.nn.BCELoss()
grad_magnitudes = []

X_data = Variable(torch.Tensor(np.array(experiment_1_data.drop('is_young_male',axis=1))))
y_data = Variable(torch.Tensor(np.array(experiment_1_outputs)))
for epoch in range(20000):
    optimizer.zero_grad()
    # Forward pass
    y_pred = mlp_pytorch(X_data)
    # Compute Loss
    loss = criterion(y_pred, y_data)
    # Backward pass
    loss.backward()
    for name, param in mlp_pytorch.named_parameters():
        if name == 'linear_1.weight':
            grad_magnitudes.append(np.abs(param.grad.numpy()).mean())

    if epoch % 1000 == 0:
        print("epoch", epoch)
        for name, param in mlp_pytorch.named_parameters():
            print(name, "value", param.data, "gradient", param.grad)
            print(name, 'value', param.data, 'gradient', param.grad)
    optimizer.step()

plt.plot(grad_magnitudes)
plt.show()
visualize_model_probs(mlp_pytorch)
```

```
/anaconda3/lib/python3.6/site-packages/torch/nn/functional.py:2016: UserWarning: Using a target si
ze (torch.Size([714])) that is different to the input size (torch.Size([714, 1])) is deprecated. P
lease ensure they have the same size.
  "Please ensure they have the same size.".format(target.size(), input.size()))
```

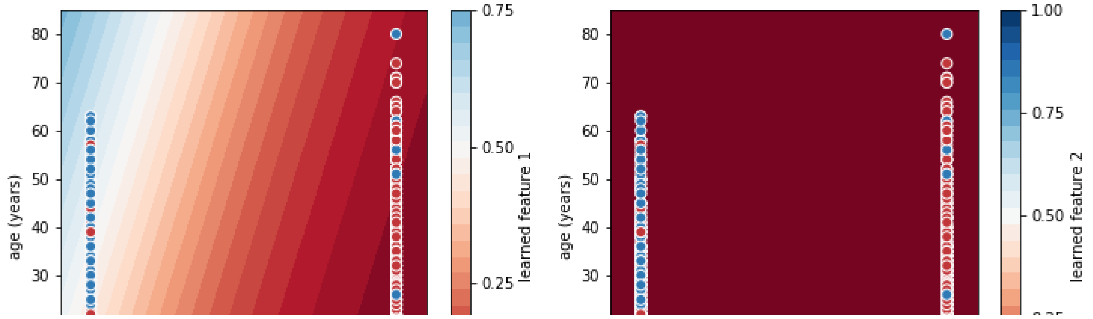
```
epoch 0
linear_1.weight value tensor([[-0.4145, -0.1487],
        [ 0.3383, -0.7046]]) gradient tensor([[0.0008, 0.0138],
        [0.0002, 0.0002]])
linear_1.bias value tensor([-0.6556,  0.2074]) gradient tensor([0.0004, 0.0003])
linear_2.weight value tensor([[ 0.4555, -0.3204]]) gradient tensor([[ 0.0005, -0.0018]])
linear_2.bias value tensor([0.3661]) gradient tensor([0.1860])
epoch 1000
linear_1.weight value tensor([[-0.6188, -0.1849],
        [ 0.3082, -0.4217]]) gradient tensor([[ 2.5671e-03, -5.5381e-03],
        [ 4.8384e-05,  2.5244e-04]])
linear_1.bias value tensor([0.5433, 0.6822]) gradient tensor([-0.0007, -0.0001])
linear_2.weight value tensor([[1.5560, 0.5713]]) gradient tensor([[-0.0009, -0.0007]])
linear_2.bias value tensor([-0.3138]) gradient tensor([0.0379])
epoch 2000
linear_1.weight value tensor([[-2.5023,  0.0119],
        [ 1.6938, -0.5345]]) gradient tensor([[ 2.2455e-02, -1.9559e-05],
        [-2.7376e-04, -6.3526e-06]])
linear_1.bias value tensor([-0.0939,  1.2680]) gradient tensor([-0.0004,  0.0002])
linear_2.weight value tensor([[3.2187, 1.5250]]) gradient tensor([[-0.0092, -0.0010]])
linear_2.bias value tensor([-1.1204]) gradient tensor([0.0476])
epoch 3000
linear_1.weight value tensor([[-3.1408,  0.0183],
        [ 3.0238, -0.4498]]) gradient tensor([[ 7.2571e-03,  7.5334e-05],
        [-2.9166e-04, -4.2530e-06]])
linear_1.bias value tensor([ 0.0436, -0.0271]) gradient tensor([-2.3927e-05,  2.9244e-04])
linear_2.weight value tensor([[4.0564, 2.2484]]) gradient tensor([[-0.0037, -0.0006]])
linear_2.bias value tensor([-1.5950]) gradient tensor([0.0156])
epoch 4000
linear_1.weight value tensor([[-3.4029,  0.0210],
        [ 3.7194, -0.4131]]) gradient tensor([[ 1.5320e-03, -2.8388e-05],
        [-1.0350e-04, -7.7651e-07]])
linear_1.bias value tensor([ 0.0461, -0.7248]) gradient tensor([6.8642e-06, 1.0096e-04])
linear_2.weight value tensor([[4.4297, 2.6086]]) gradient tensor([[-0.0009, -0.0001]])
linear_2.bias value tensor([-1.7965]) gradient tensor([0.0029])
epoch 5000
linear_1.weight value tensor([[-3.4792,  0.0210],
        [ 4.0459, -0.4076]]) gradient tensor([[ 2.3661e-04, -5.1948e-05],
        [-3.3050e-05,  1.5887e-08]])
linear_1.bias value tensor([ 0.0203, -1.0133]) gradient tensor([1.0550e-05, 2.1519e-05])
linear_2.weight value tensor([[4.5712, 2.6741]]) gradient tensor([[-2.4139e-04,  3.7370e-06]])
linear_2.bias value tensor([-1.8332]) gradient tensor([-7.5017e-05])
epoch 6000
linear_1.weight value tensor([[-3.5063,  0.0204],
        [ 4.2207, -0.4152]]) gradient tensor([[ 9.8733e-05, -2.3553e-04],
        [-1.1252e-05, -2.6655e-07]])
linear_1.bias value tensor([-0.0251, -1.0812]) gradient tensor([3.2779e-06, 4.7078e-07])
linear_2.weight value tensor([[4.6558, 2.6409]]) gradient tensor([[-1.3632e-04,  1.0659e-05]])
linear_2.bias value tensor([-1.8185]) gradient tensor([-0.0002])
epoch 7000
linear_1.weight value tensor([[-3.5322,  0.0200],
        [ 4.3046, -0.4229]]) gradient tensor([[ 6.5952e-05,  3.6335e-05],
        [-2.2271e-06,  1.1456e-07]])
linear_1.bias value tensor([-0.0779, -1.0685]) gradient tensor([ 8.0868e-06, -9.9563e-07])
linear_2.weight value tensor([[4.7426, 2.6048]]) gradient tensor([[-8.4636e-05,  3.9071e-06]])
linear_2.bias value tensor([-1.7997]) gradient tensor([-0.0001])
epoch 8000
linear_1.weight value tensor([[-3.5590,  0.0196],
        [ 4.3146, -0.4257]]) gradient tensor([[ 3.7913e-05, -3.9865e-05],
        [ 4.4981e-07, -9.3525e-08]])
linear_1.bias value tensor([-0.1290, -1.0528]) gradient tensor([ 3.3678e-06, -3.7719e-07])
linear_2.weight value tensor([[4.8315, 2.5885]]) gradient tensor([[-5.1868e-05,  7.6236e-07]])
linear_2.bias value tensor([-1.7829]) gradient tensor([-6.2459e-05])
epoch 9000
linear_1.weight value tensor([[-3.5817,  0.0191],
linear_1.weight value tensor([[-3.5817,  0.0191],
        [ 4.2947, -0.4257]]) gradient tensor([[ 6.4422e-06, -5.0363e-03],
        [-5.9151e-07, -1.2816e-05]])
linear_1.bias value tensor([-0.1732, -1.0448]) gradient tensor([-1.5123e-04, -1.5579e-06])
linear_2.weight value tensor([[4.9158, 2.5835]]) gradient tensor([[-1.0626e-04, -9.5961e-07]])
linear_2.bias value tensor([-1.7695]) gradient tensor([-0.0002])
epoch 10000
linear_1.weight value tensor([[-3.5938,  0.0189],
        [ 4.2800, -0.4254]]) gradient tensor([[ 2.1533e-06, -4.3441e-05],
        [ 1.0192e-07, -1.1148e-07]])
linear_1.bias value tensor([-0.2077, -1.0423]) gradient tensor([ 8.4356e-07, -1.9088e-08])
linear_2.weight value tensor([[4.9900, 2.5811]]) gradient tensor([[-1.5673e-05,  5.6170e-08]])
linear_2.bias value tensor([-1.7607]) gradient tensor([-1.0403e-05])
epoch 11000
linear_1.weight value tensor([[-3.5881,  0.0186],
        [ 4.2759, -0.4253]]) gradient tensor([[-7.4118e-06, -1.1230e-03],
        [-2.1938e-07, -2.7583e-06]])
linear_1.bias value tensor([-0.2327, -1.0436]) gradient tensor([-3.2973e-05, -3.0063e-07])
linear_2.weight value tensor([[5.0541, 2.5797]]) gradient tensor([[-2.4662e-05, -2.3692e-07]])
linear_2.bias value tensor([-1.7574]) gradient tensor([-3.5731e-05])
epoch 12000
linear_1.weight value tensor([[-3.5593,  0.0182],
        [ 4.2807, -0.4254]]) gradient tensor([[-1.3927e-05, -3.3873e-03],
        [-7.1421e-07, -8.4721e-06]])
linear_1.bias value tensor([-0.2516, -1.0484]) gradient tensor([-1.0251e-04, -9.5566e-07])
linear_2.weight value tensor([[5.1172, 2.5796]]) gradient tensor([[-5.3764e-05, -8.1916e-07]])
linear_2.bias value tensor([-1.7597]) gradient tensor([-0.0001])
epoch 13000
linear_1.weight value tensor([[-3.5050,  0.0179],
        [ 4.2930, -0.4255]]) gradient tensor([[-6.2939e-06,  1.1384e-03],
        [ 1.8415e-07,  2.8340e-06]])
linear_1.bias value tensor([-0.2709, -1.0568]) gradient tensor([3.6698e-05, 3.6591e-07])
linear_2.weight value tensor([[5.1976, 2.5805]]) gradient tensor([[9.5577e-06, 2.7235e-07]])
linear_2.bias value tensor([-1.7672]) gradient tensor([3.7914e-05])
epoch 14000
linear_1.weight value tensor([[-3.4257,  0.0174],
        [ 4.3116, -0.4257]]) gradient tensor([[-6.9060e-06, -1.1790e-04],
        [-6.2311e-08, -2.9505e-07]])
linear_1.bias value tensor([-0.2963, -1.0688]) gradient tensor([-1.0516e-06, -1.2365e-08])
linear_2.weight value tensor([[5.3113, 2.5821]]) gradient tensor([[-8.1146e-06, -4.1779e-08]])
linear_2.bias value tensor([-1.7795]) gradient tensor([-4.0163e-07])
epoch 15000
linear_1.weight value tensor([[-3.3293,  0.0168],
        [ 4.3340, -0.4259]]) gradient tensor([[-4.5459e-06,  6.7838e-06],
        [-2.4555e-08,  2.5520e-08]])
linear_1.bias value tensor([-0.3286, -1.0832]) gradient tensor([3.5741e-06, 2.1646e-08])
linear_2.weight value tensor([[5.4605, 2.5842]]) gradient tensor([[-7.2536e-06, -7.0722e-09]])
linear_2.bias value tensor([-1.7954]) gradient tensor([2.6632e-06])
epoch 16000
linear_1.weight value tensor([[-3.2315,  0.0162],
        [ 4.3559, -0.4261]]) gradient tensor([[-2.6695e-06,  1.0886e-05],
        [-1.6815e-08,  2.6514e-08]])
linear_1.bias value tensor([-0.3652, -1.0975]) gradient tensor([4.2199e-06, 2.0574e-08])
linear_2.weight value tensor([[5.6316, 2.5864]]) gradient tensor([[-7.5754e-06, -8.5856e-10]])
linear_2.bias value tensor([-1.8125]) gradient tensor([2.2664e-06])
epoch 17000
linear_1.weight value tensor([[-3.1430,  0.0156],
        [ 4.3748, -0.4262]]) gradient tensor([[-1.8965e-06, -1.8765e-04],
        [-5.5379e-08, -4.5043e-07]])
linear_1.bias value tensor([-0.4026, -1.1102]) gradient tensor([-1.5810e-06, -4.1339e-08])
linear_2.weight value tensor([[5.8086, 2.5883]]) gradient tensor([[-9.6317e-06, -6.1147e-08]])
linear_2.bias value tensor([-1.8286]) gradient tensor([-3.3493e-06])
epoch 18000
linear_1.weight value tensor([[-3.0668,  0.0151],
        [ 4.3905, -0.4263]]) gradient tensor([[-8.9720e-07,  1.4407e-05],
        [-9.5462e-09,  4.0382e-08]])
linear_1.bias value tensor([-0.4384, -1.1208]) gradient tensor([4.5546e-06, 1.5097e-08])
linear_2.weight value tensor([[5.9807, 2.5899]]) gradient tensor([[-7.1521e-06,  4.3947e-09]])
linear_2.bias value tensor([-1.8431]) gradient tensor([1.8266e-06])
epoch 19000
linear_1.weight value tensor([[-3.0013,  0.0147],
        [ 4.4035, -0.4264]]) gradient tensor([[1.5571e-08, 3.1853e-04],
        [5.1470e-08, 6.7560e-07]])
linear_1.bias value tensor([-0.4717, -1.1298]) gradient tensor([1.3555e-05, 9.5572e-08])
linear_2.weight value tensor([[6.1446, 2.5912]]) gradient tensor([[-3.8617e-06,  9.1022e-08]])
linear_2.bias value tensor([-1.8559]) gradient tensor([9.3128e-06])
```

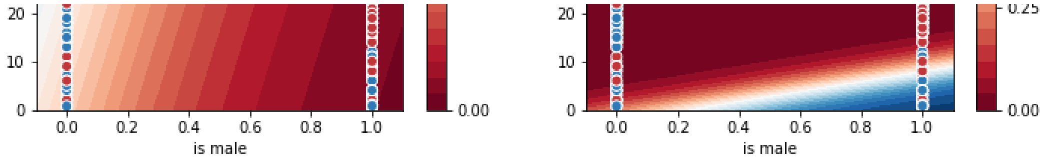




In [11]:

```
xx, yy = np.mgrid[-.1:1.1:.01, 0:85:.1]
grid = np.c_[xx.ravel(), yy.ravel()]

f = plt.figure(figsize=(12, 5))
hidden_units = mlp_pytorch.hidden(Variable(torch.Tensor(grid))).detach().numpy()
for i in range(2):
    ax = f.add_subplot(1,2,i+1)
    contour = ax.contourf(xx, yy, hidden_units[:,i].reshape(xx.shape), 25, cmap="RdBu",
                          vmin=0, vmax=1)

    ax.scatter(experiment_1_data['male'], experiment_1_data['Age'], c=experiment_1_outputs, s=50,
               cmap="RdBu", vmin=-.2, vmax=1.2,
               edgecolor="white", linewidth=1)
    ax_c = f.colorbar(contour)
    ax_c.set_label("learned feature %d" %(i+1))
    ax_c.set_ticks([0, .25, .5, .75, 1])

    ax.set(xlim=(-.1, 1.1),
        ylim=(0, 85),
        xlabel="is male", ylabel="age (years)")
plt.show()
```




20
10
0
0.0
0.2
0.4
0.6
0.8
1.0
is male
0.00
20
10
0
0.0
0.2
0.4
0.6
0.8
1.0
is male
0.25
0.00

In [0]: