Problem Set 4: Neural Networks

This assignment requires a working IPython Notebook installation, which you should already have. If not, please refer to the instructions in Problem Set 2.

The programming part is adapted from [Stanford CS231n (http://cs231n.stanford.edu/)](http://cs231n.stanford.edu/).

In part 2 (programming) of this assignment, you DO NOT need to make any modification code in this IPython Notebook. Instead you will implement your own simple neural network in the mlp.py file. Please attach your written solutions for part 1 and part 3 in this IPython Notebook.

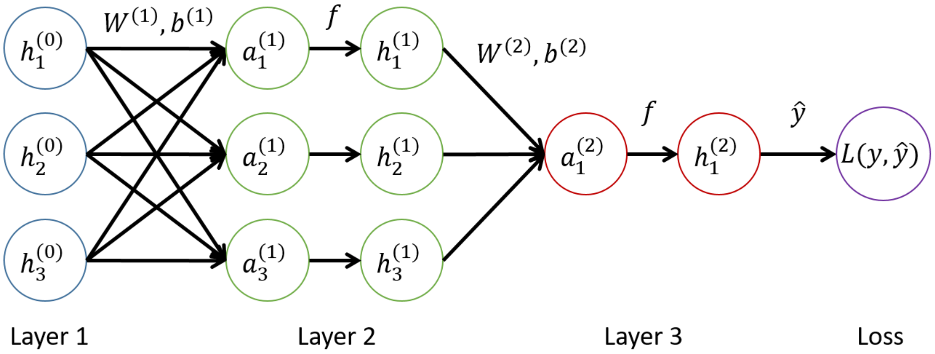
Total: 100 points.

[30pts] Problem 1: Backprop in a simple MLP

This problem asks you to derive all the steps of the backpropagation algorithm for a simple classification network. Consider a fully-connected neural network, also known as a multi-layer perceptron (MLP), with a single hidden layer and a one-node output layer. The hidden and output nodes use an elementwise sigmoid activation function and the loss layer uses cross-entropy loss:

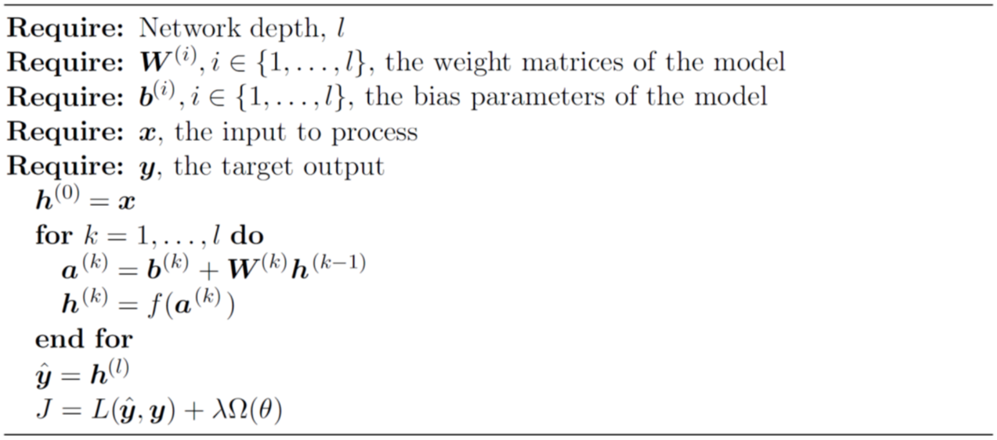
$f(z) = \frac{1}{1+exp(-z))}$
$L(\hat{y}, y) = -yln(\hat{y}) - (1-y)ln(1-\hat{y})$

The computation graph for an example network is shown below. Note that it has an equal number of nodes in the input and hidden layer (3 each), but, in general, they need not be equal. Also, to make the application of backprop easier, we show the *computation graph* which shows the dot product and activation functions as their own nodes, rather than the usual graph showing a single node for both.



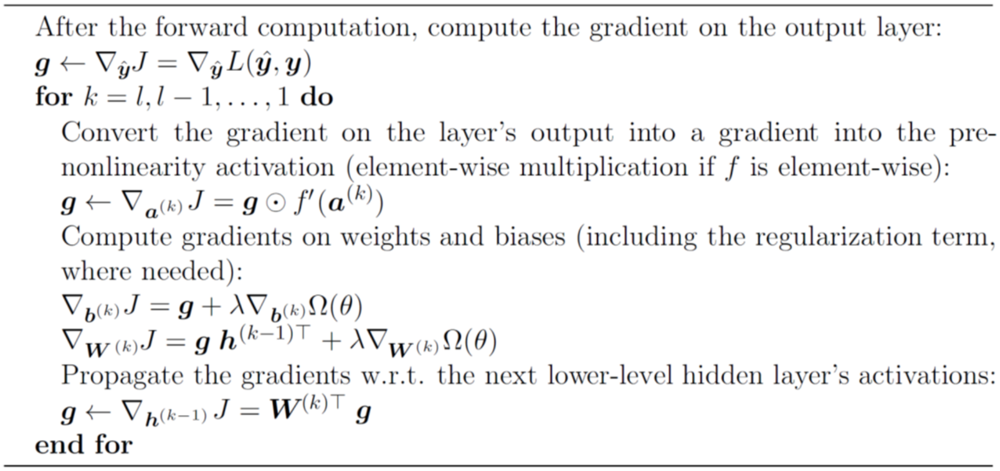
The forward and backward computation are given below. NOTE: We assume no regularization, so you can omit the terms involving Ω.

The forward step is:



```
Require: Network depth, $l$
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \dots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \dots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output
  $\boldsymbol{h}^{(0)} = \boldsymbol{x}$
  for $k = 1, \dots, l$ do
    $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k-1)}$
    $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
  end for
  $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
  $J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda\Omega(\theta)$
```

and the backward step is:



```
After the forward computation, compute the gradient on the output layer:
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for $k = l, l-1, \dots, 1$ do
  Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if $f$ is element-wise):
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
  Compute gradients on weights and biases (including the regularization term, where needed):
  $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$
  $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$
  Propagate the gradients w.r.t. the next lower-level hidden layer's activations:
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$
end for
```

Write down each step of the backward pass explicitly for all layers, i.e. for the loss and $k = 2, 1$, compute all gradients above, expressing them as a function of variables x, y, h, W, b. We start by giving an example. Note that we have replaced the superscript notation $u^{(i)}$ with u^i, and $\odot$ stands for element-wise multiplication.

$$\nabla_{\hat{y}} L(\hat{y}, y) = \nabla_{\hat{y}}[-yln(\hat{y}) - (1-y)ln(1-\hat{y})] = \frac{\hat{y}-y}{(1-\hat{y})\hat{y}} = \frac{h^2-y}{(1-h^2)h^2}$$

Next, please derive the following.

Hint: you should substitute the updated values for the gradient g *in each step and simplify as much as possible.*

[5pts] Q1.1: $\nabla_{a^2} \boldsymbol{J}$

$$\nabla_{a^2} \boldsymbol{J} = \frac{\partial \boldsymbol{J}}{\partial h^2} \cdot \frac{\partial h^2}{\partial a^2} = \nabla_{\hat{y}} L(\hat{y}, y) \cdot f'(a^2) = \frac{h^2 - y}{(1-h^2)h^2} \cdot h^2(1-h^2) = h^2 - y$$

We note here that $f'(x) = f(x) * (1 - f(x))$ where $f(x)$ is the sigmoid function.We now update g to be equal to the obtaine value, $g = h^2 - y$

[5pts] Q1.2: $\nabla_{b^2} \boldsymbol{J}$

$$\nabla_{b^2} J = \frac{\partial J}{\partial h^2} \cdot \frac{\partial h^2}{\partial a^2} \cdot \frac{\partial a^2}{\partial b^2} = g \cdot 1 = g = h^2 - y$$

[5pts] Q1.3: $\nabla_{W^2} J$
Hint: this should be a vector, since W^2 is a vector.

$$\nabla_{W^2} J = \frac{\partial J}{\partial h^2} \cdot \frac{\partial h^2}{\partial a^2} \cdot \frac{\partial a^2}{\partial W^2} = g \cdot {h^1}^T = (h^2 - y) \cdot {h^1}^T$$

[5pts] Q1.4: $\nabla_{h^1} J$

$$\nabla_{h^1} J = \frac{\partial J}{\partial h^2} \cdot \frac{\partial h^2}{\partial a^2} \cdot \frac{\partial a^2}{\partial h^1} = {W^2}^T \cdot g = {W^2}^T \cdot (h^2 - y)$$

We now proceed to update the value of g by using the above value, hence now $g = {W^2}^T \cdot (h^2 - y)$

[5pts] Q1.5: $\nabla_{b^1} J$, $\nabla_{W^1} J$

First we update g by taking the gradient of J wrt a^1

$$\nabla_{a^1} J = \frac{\partial J}{\partial h^2} \cdot \frac{\partial h^2}{\partial a^2} \cdot \frac{\partial a^2}{\partial h^1} \cdot \frac{\partial h^1}{\partial a^1} = g \cdot f'(a^1) = {W^2}^T \cdot (h^2 - y) \cdot h^1 \cdot (1 - h^1)$$

Hence we update the value of g to be $g = {W^2}^T \cdot (h^2 - y) \cdot h^1 \cdot (1 - h^1)$

We now proceed to calculate the gradients with respect to b^1 and W^1.

- $\nabla_{b^1} J = \frac{\partial J}{\partial h^2} \cdot \frac{\partial h^2}{\partial a^2} \cdot \frac{\partial a^2}{\partial h^1} \cdot \frac{\partial h^1}{\partial a^1} \cdot \frac{\partial a^1}{\partial b^1} = g \cdot 1 = g = {W^2}^T \cdot (h^2 - y) \cdot h^1 \cdot (1 - h^1)$
- $\nabla_{W^1} J = \frac{\partial J}{\partial h^2} \cdot \frac{\partial h^2}{\partial a^2} \cdot \frac{\partial a^2}{\partial h^1} \cdot \frac{\partial h^1}{\partial a^1} \cdot \frac{\partial a^1}{\partial W^1} = g \cdot {h^0}^T = {W^2}^T \cdot (h^2 - y) \cdot h^1 \cdot (1 - h^1) \cdot {h^0}^T$

[5pts] Q1.6 Briefly, explain how the computational speed of backpropagation would be affected if it did not include a forward pass

Without the forward pass, during our backpropogation step we would not have access to the values we need to compute the derivatives at each step. This would mean that we would have to compute them each time for each backpropagation step. Additionaly, we would not be able to reuse the values since they change at each layer. As a result without the forward pass we would have to perform numerous time-consuming computations if our network was larger than the one we have in this example. As a result it would take much longer to run the neural network. Storing the variables in the forward pass requires more memory but the computation time will be significantly reduced.

[50pts] Problem 2 (Programming): Implementing a simple MLP

In this problem we will develop a neural network with fully-connected layers, or Multi-Layer Perceptron (MLP). We will use it in classification tasks.

In the current directory, you can find a file `mlp.py` , which contains the definition for class `TwoLayerMLP` . As the name suggests, it implements a 2-layer MLP, or MLP with 1 *hidden* layer. You will implement your code in the same file, and call the member functions in this notebook. Below is some initialization. The `autoreload` command makes sure that `mlp.py` is periodically reloaded.

In [1]:

```
# setup
import numpy as np
import matplotlib.pyplot as plt
from mlp import TwoLayerMLP

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-i
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y)))
```

Next we initialize a toy model and some toy data, the task is to classify five 4-d vectors.

In [2]:

```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.
input_size = 4
hidden_size = 10
num_classes = 3
num_inputs = 5

def init_toy_model(actv, std=1e-1):
    np.random.seed(0)
    return TwoLayerMLP(input_size, hidden_size, num_classes, std=std, activ

def init_toy_data():
    np.random.seed(1)
    X = 10 * np.random.randn(num_inputs, input_size)
    y = np.array([0, 1, 2, 2, 1])
    return X, y

X, y = init_toy_data()
print('X = ', X)
print()
print('y = ', y)
```

```
X =  [[ 16.24345364  -6.11756414  -5.28171752 -10.72968622]
 [  8.65407629 -23.01538697  17.44811764  -7.61206901]
 [  3.19039096  -2.49370375  14.62107937 -20.60140709]
 [ -3.22417204  -3.84054355  11.33769442 -10.99891267]
 [ -1.72428208  -8.77858418   0.42213747   5.82815214]]

y =  [0 1 2 2 1]
```

[5pts] Q2.1 Forward pass: Sigmoid

Our 2-layer MLP uses a softmax output layer (**note**: this means that you don't need to apply a sigmoid on the output) and the multiclass cross-entropy loss to perform classification. Both are defined in Problem Set 2.

Softmax function:
For class j:

$$P(y = j|x) = \frac{\exp(z_j)}{\sum_{k=1}^{K} \exp(z_k)}$$

Multiclass cross-entropy loss function:
y - binary indicator (0 or 1) if class label c is the correct classification

$$J = \frac{1}{m} \sum_{i=1}^{m} \sum_{c=1}^{C} [\ - y_{(c)} log(P(y_{(c)}|x^{(i)}))\ ]$$

Please take a look at method `TwoLayerMLP.loss` in the file `mlp.py` . This function takes in the data and weight parameters, and computes the class scores (aka logits), the loss L, and the gradients on the parameters.

- Complete the implementation of forward pass (up to the computation of `scores`) for the sigmoid activation: $\sigma(x) = \frac{1}{1+exp(-x)}$.

Note 1: Softmax cross entropy loss involves the [log-sum-exp operation (https://en.wikipedia.org/wiki/LogSumExp)](https://en.wikipedia.org/wiki/LogSumExp). This can result in numerical underflow/overflow. Read about the solution in the link, and try to understand the calculation of `loss` in the code.

Note 2: You're strongly encouraged to implement in a vectorized way and avoid using slower `for` loops. Note that most numpy functions support vector inputs.

Check the correctness of your forward pass below. The difference should be very small (<1e-6).

In [3]:

```
net = init_toy_model('sigmoid')
loss, _ = net.loss(X, y, reg=0.1)
correct_loss = 1.182248
print(loss)
print('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss)))
```

```
1.1822479803941373
Difference between your loss and correct loss:
1.9605862711102873e-08
```

[10pts] Q2.2 Backward pass: Sigmoid

- For sigmoid activation, complete the computation of `grads` , which stores the gradient of the loss with respect to the variables `W1` , `b1` , `W2` , and `b2` .

Now debug your backward pass using a numeric gradient check. Again, the differences should be very small.

In [4]:

```
# Use numeric gradient checking to check your implementation of the backwar
# If your implementation is correct, the difference between the numeric and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b
from utils import eval_numerical_gradient

loss, grads = net.loss(X, y, reg=0.1)

# these should all be very small
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.1)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name], ver
    print('%s max relative error: %e'%(param_name, rel_error(param_grad_num
```

```
W2 max relative error: 8.048892e-10
b2 max relative error: 5.553999e-11
W1 max relative error: 1.126755e-08
b1 max relative error: 2.035406e-06
```

[5pts] Q2.3 Train the Sigmoid network

To train the network we will use stochastic gradient descent (SGD), implemented in `TwoLayerNet.train` . Then we train a two-layer network on toy data.

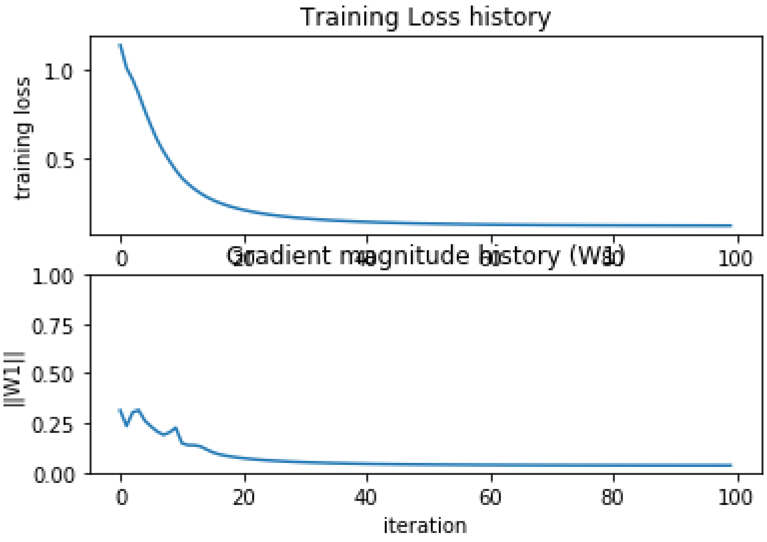
- Implement the prediction function `TwoLayerNet.predict` , which is called during training to keep track of training and validation accuracy.

You should get the final training loss around 0.1, which is good, but not too great for such a toy problem. One problem is that the gradient magnitude for W1 (the first layer weights) stays small all the time, and the neural net doesn't get much "learning signals". This has to do with the saturation problem of the sigmoid activation function.

In [5]:

```
net = init_toy_model('sigmoid', std=1e-1)
stats = net.train(X, y, X, y,
                  learning_rate=0.5, reg=1e-5,
                  num_epochs=100, verbose=False)
print('Final training loss: ', stats['loss_history'][-1])

# plot the loss history and gradient magnitudes
plt.subplot(2, 1, 1)
plt.plot(stats['loss_history'])
plt.xlabel('epoch')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.subplot(2, 1, 2)
plt.plot(stats['grad_magnitude_history'])
plt.xlabel('iteration')
plt.ylabel('||W1||')
plt.ylim(0,1)
plt.title('Gradient magnitude history (W1)')
plt.show()
```

```
Final training loss:  0.10926794610680679
```



[5pts] Q2.4 Using ReLU activation

The Rectified Linear Unit (ReLU) activation is also widely used: $ReLU(x) = max(0, x)$.

- Complete the implementation for the ReLU activation (forward and backward) in `mlp.py` .
- Train the network with ReLU, and report your final training loss.

Make sure you first pass the numerical gradient check on toy data.

In [6]:
```
net = init_toy_model('relu', std=1e-1)

loss, grads = net.loss(X, y, reg=0.1)
print('loss = ', loss)  # correct_loss = 1.320973

# The differences should all be very small
print('checking gradients')
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.1)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name], ver
    print('%s max relative error: %e'%(param_name, rel_error(param_grad_num
```

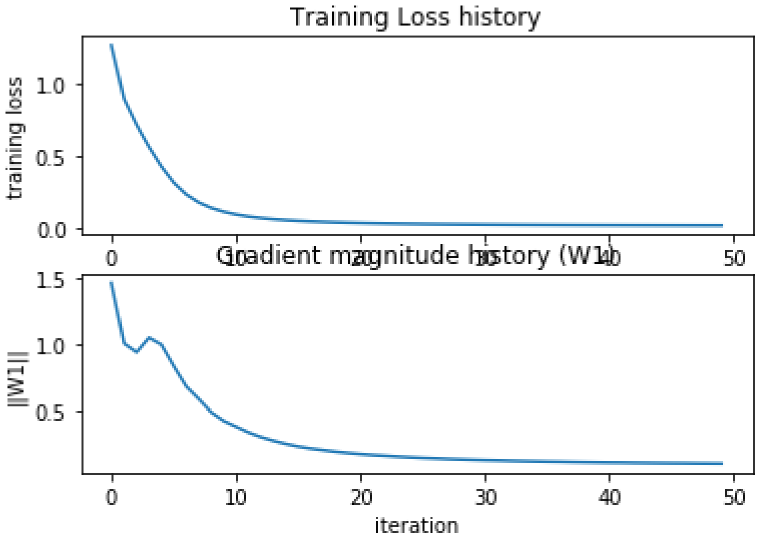
```
loss =  1.3037878913298206
checking gradients
W2 max relative error: 3.440708e-09
b2 max relative error: 3.865091e-11
W1 max relative error: 3.561318e-09
b1 max relative error: 8.994864e-10
```

Now that it's working, let's train the network. Does the net get stronger learning signals (i.e. gradients) this time? Report your final training loss.

In [7]:

```
net = init_toy_model('relu', std=1e-1)
stats = net.train(X, y, X, y,
                  learning_rate=0.1, reg=1e-5,
                  num_epochs=50, verbose=False)

print('Final training loss: ', stats['loss_history'][-1])

# plot the loss history
plt.subplot(2, 1, 1)
plt.plot(stats['loss_history'])
plt.xlabel('epoch')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.subplot(2, 1, 2)
plt.plot(stats['grad_magnitude_history'])
plt.xlabel('iteration')
plt.ylabel('||W1||')
plt.title('Gradient magnitude history (W1)')
plt.show()
```

```
Final training loss:  0.0178562204869839
```



Load MNIST data

Now that you have implemented a two-layer network that works on toy data, let's try some real data. The MNIST dataset is a standard machine learning benchmark. It consists of 70,000 grayscale handwritten digit images, which we split into 50,000 training, 10,000 validation and 10,000 testing. The images are of size 28x28, which are flattened into 784-d vectors.

Note 1: the function `get_MNIST_data` requires the `scikit-learn` package. If you previously did anaconda installation to set up your Python environment, you should already have it. Otherwise, you can install it following the instructions here: http://scikit-learn.org/stable/install.html (http://scikit-learn.org/stable/install.html)

Note 2: If you encounter a `HTTP 500` error, that is likely temporary, just try again.

Note 3: Ensure that the downloaded MNIST file is 55.4MB (smaller file-sizes could indicate an incomplete download - which is possible)

In [8]:

```
# load MNIST
from utils import get_MNIST_data
X_train, y_train, X_val, y_val, X_test, y_test = get_MNIST_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
/Users/danielvaroli/anaconda3/lib/python3.7/site-packages/sklearn/utils/d
eprecation.py:85: DeprecationWarning: Function fetch_mldata is deprecate
d; fetch_mldata was deprecated in version 0.20 and will be removed in ver
sion 0.22. Please use fetch_openml.
  warnings.warn(msg, category=DeprecationWarning)
/Users/danielvaroli/anaconda3/lib/python3.7/site-packages/sklearn/utils/d
eprecation.py:85: DeprecationWarning: Function mldata_filename is depreca
ted; mldata_filename was deprecated in version 0.20 and will be removed i
n version 0.22. Please use fetch_openml.
  warnings.warn(msg, category=DeprecationWarning)

Train data shape:  (50000, 784)
Train labels shape:  (50000,)
Validation data shape:  (10000, 784)
Validation labels shape:  (10000,)
Test data shape:  (10000, 784)
Test labels shape:  (10000,)
```

Q2.5 Train a network on MNIST

We will now train a network on MNIST with 64 hidden units in the hidden layer. We train it using SGD, and decrease the learning rate with an exponential rate over time; this is achieved by multiplying the learning rate with a constant factor `learning_rate_decay` (which is less than 1) after each epoch. In effect, we are using a high learning rate initially, which is good for exploring the solution space, and using lower learning rates later to encourage convergence to a local minimum (or saddle point (http://www.offconvex.org/2016/03/22/saddlepoints/), which may happen more often).

- Train your MNIST network with 2 different activation functions: sigmoid and ReLU.

We first define some variables and utility functions. The `plot_stats` function plots the histories of gradient magnitude, training loss, and accuracies on the training and validation sets. The `visualize_weights` function visualizes the weights learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized. Both functions help you to diagnose the training process.

In [9]:

```
input_size = 28 * 28
hidden_size = 64
num_classes = 10

# Plot the loss function and train / validation accuracies
def plot_stats(stats):
    plt.subplot(3, 1, 1)
    plt.plot(stats['grad_magnitude_history'])
    plt.title('Gradient magnitude history (W1)')
    plt.xlabel('Iteration')
    plt.ylabel('||W1||')
    plt.ylim(0, np.minimum(100,np.max(stats['grad_magnitude_history'])))
    plt.subplot(3, 1, 2)
    plt.plot(stats['loss_history'])
    plt.title('Loss history')
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.ylim(0, 100)
    plt.subplot(3, 1, 3)
    plt.plot(stats['train_acc_history'], label='train')
    plt.plot(stats['val_acc_history'], label='val')
    plt.title('Classification accuracy history')
    plt.xlabel('Epoch')
    plt.ylabel('Clasification accuracy')
    plt.show()

# Visualize the weights of the network
from utils import visualize_grid
def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.T.reshape(-1, 28, 28)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()
```

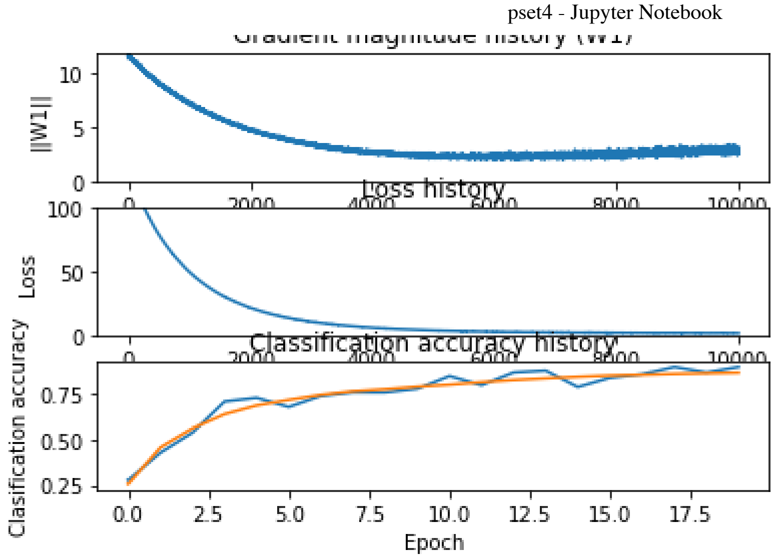
[10pts] Q2.5.1 Sigmoid network

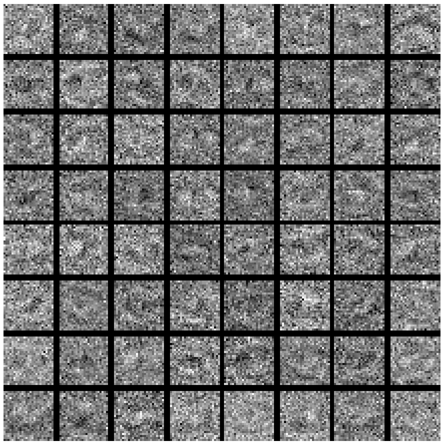
In [10]:

```
sigmoid_net = TwoLayerMLP(input_size, hidden_size, num_classes, activation=

# Train the network
sigmoid_stats = sigmoid_net.train(X_train, y_train, X_val, y_val,
                                  num_epochs=20, batch_size=100,
                                  learning_rate=1e-3,  learning_rate_decay=
                                  reg=0.5, verbose=True)

# Predict on the training set
train_acc = (sigmoid_net.predict(X_train) == y_train).mean()
print('Sigmoid final training accuracy: ', train_acc)

# Predict on the validation set
val_acc = (sigmoid_net.predict(X_val) == y_val).mean()
print('Sigmoid final validation accuracy: ', val_acc)

# Predict on the test set
test_acc = (sigmoid_net.predict(X_test) == y_test).mean()
print('Sigmoid test accuracy: ', test_acc)

# show stats and visualizations
plot_stats(sigmoid_stats)
show_net_weights(sigmoid_net)
```

```
Epoch 1: loss 78.980750, train_acc 0.280000, val_acc 0.258300
Epoch 2: loss 49.838630, train_acc 0.430000, val_acc 0.459300
Epoch 3: loss 32.431795, train_acc 0.540000, val_acc 0.563200
Epoch 4: loss 21.732892, train_acc 0.710000, val_acc 0.640700
Epoch 5: loss 15.111877, train_acc 0.730000, val_acc 0.688400
Epoch 6: loss 10.906376, train_acc 0.680000, val_acc 0.718700
Epoch 7: loss 8.077669, train_acc 0.740000, val_acc 0.746200
Epoch 8: loss 6.229287, train_acc 0.760000, val_acc 0.766000
Epoch 9: loss 4.979639, train_acc 0.760000, val_acc 0.777300
Epoch 10: loss 4.108814, train_acc 0.780000, val_acc 0.791700
Epoch 11: loss 3.480617, train_acc 0.850000, val_acc 0.802500
Epoch 12: loss 3.065671, train_acc 0.800000, val_acc 0.816400
Epoch 13: loss 2.673266, train_acc 0.870000, val_acc 0.828100
Epoch 14: loss 2.496009, train_acc 0.880000, val_acc 0.837300
Epoch 15: loss 2.355747, train_acc 0.790000, val_acc 0.845900
Epoch 16: loss 2.138493, train_acc 0.840000, val_acc 0.852900
Epoch 17: loss 2.090350, train_acc 0.860000, val_acc 0.856900
Epoch 18: loss 1.974509, train_acc 0.900000, val_acc 0.861900
Epoch 19: loss 1.952709, train_acc 0.870000, val_acc 0.865300
Epoch 20: loss 1.859285, train_acc 0.900000, val_acc 0.868600
Sigmoid final training accuracy:  0.87384
Sigmoid final validation accuracy:  0.8686
Sigmoid test accuracy:  0.8676
```

Gradient magnitude history (W1)



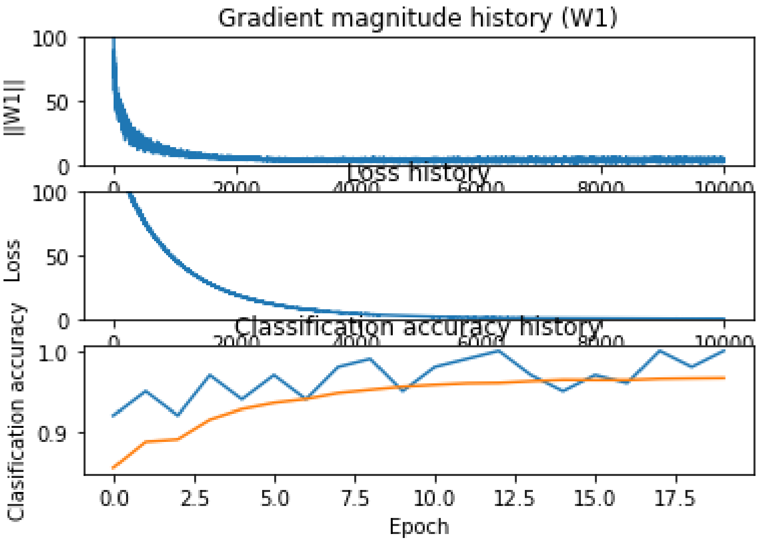


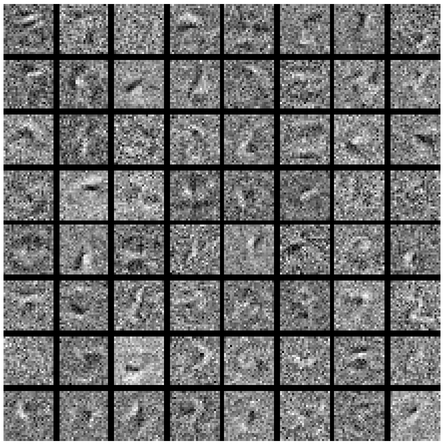
[10pts] Q2.5.2 ReLU network

In [11]:

```
relu_net = TwoLayerMLP(input_size, hidden_size, num_classes, activation='re

# Train the network
relu_stats = relu_net.train(X_train, y_train, X_val, y_val,
                            num_epochs=20, batch_size=100,
                            learning_rate=1e-3, learning_rate_decay=0.95,
                            reg=0.5, verbose=True)
# Predict on the training set
train_acc = (relu_net.predict(X_train) == y_train).mean()
print('ReLU final training accuracy: ', train_acc)

# Predict on the validation set
val_acc = (relu_net.predict(X_val) == y_val).mean()
print('ReLU final validation accuracy: ', val_acc)

# Predict on the test set
test_acc = (relu_net.predict(X_test) == y_test).mean()
print('ReLU test accuracy: ', test_acc)

# show stats and visualizations
plot_stats(relu_stats)
show_net_weights(relu_net)
```

```
Epoch 1: loss 76.058480, train_acc 0.920000, val_acc 0.855500
Epoch 2: loss 46.872686, train_acc 0.950000, val_acc 0.887600
Epoch 3: loss 29.964734, train_acc 0.920000, val_acc 0.890600
Epoch 4: loss 19.383910, train_acc 0.970000, val_acc 0.915000
Epoch 5: loss 13.070584, train_acc 0.940000, val_acc 0.928000
Epoch 6: loss 8.888005, train_acc 0.970000, val_acc 0.935600
Epoch 7: loss 6.325541, train_acc 0.940000, val_acc 0.940300
Epoch 8: loss 4.410171, train_acc 0.980000, val_acc 0.947700
Epoch 9: loss 3.229262, train_acc 0.990000, val_acc 0.951700
Epoch 10: loss 2.439701, train_acc 0.950000, val_acc 0.955500
Epoch 11: loss 1.817755, train_acc 0.980000, val_acc 0.957800
Epoch 12: loss 1.411403, train_acc 0.990000, val_acc 0.959700
Epoch 13: loss 1.120687, train_acc 1.000000, val_acc 0.960200
Epoch 14: loss 0.994314, train_acc 0.970000, val_acc 0.962300
Epoch 15: loss 0.822748, train_acc 0.950000, val_acc 0.964000
Epoch 16: loss 0.714908, train_acc 0.970000, val_acc 0.963800
Epoch 17: loss 0.586254, train_acc 0.960000, val_acc 0.963800
Epoch 18: loss 0.474870, train_acc 1.000000, val_acc 0.965300
Epoch 19: loss 0.476649, train_acc 0.980000, val_acc 0.965800
Epoch 20: loss 0.379714, train_acc 1.000000, val_acc 0.966200
ReLU final training accuracy:  0.9735
ReLU final validation accuracy:  0.9662
ReLU test accuracy:  0.9632
```





[5pts] Q2.5.3

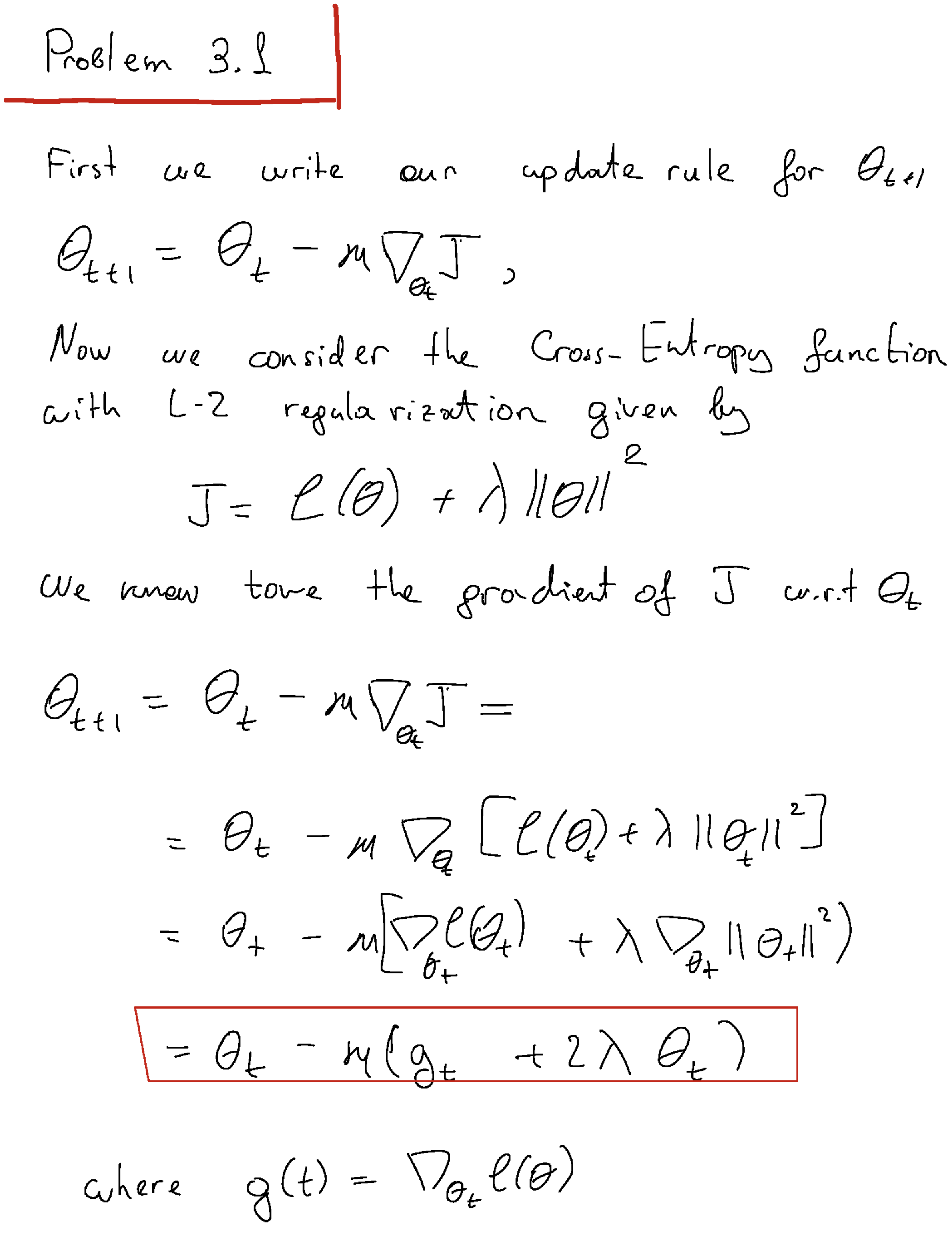
Based on the outputs of the function call above, we can see that the test accuracy of the ReLU method is around 97%, however the test accuracy of the sigmoid activation function is around 87%, which is higher, hence I would pick ReLU. In addition the ReLU method is sparse, which means that it allows a network to easily obtain sparse representations.Furthermore, with the ReLU the chances of the occurance of vanishing grqdients is reduced.

[20pts] Problem 3: Simple Regularization Methods

You may have noticed the `reg` parameter in `TwoLayerMLP.loss` , controlling "regularization strength". In learning neural networks, aside from minimizing a loss function $\mathcal{L}(\theta)$ with respect to the network parameters θ, we usually explicitly or implicitly add some regularization term to reduce overfitting. A simple and popular regularization strategy is to penalize some *norm* of θ.

[10pts] Q3.1: L2 regularization

We can penalize the L2 norm of θ: we modify our objective function to be $\mathcal{L}(\theta) + \lambda \|\theta\|^2$ where λ is the weight of regularization. We will minimize this objective using gradient descent with step size η. Derive the update rule: at time $t + 1$, express the new parameters θ_{t+1} in terms of the old parameters θ_t, the gradient $g_t = \frac{\partial \mathcal{L}}{\partial \theta_t}$, η, and λ.



Problem 3.1

First we write our update rule for θ_{t+1}

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta_t} J ,$$

Now we consider the Cross-Entropy function with L-2 regularization given by

$$J = \ell(\theta) + \lambda ||\theta||^2$$

We now tore the gradient of J w.r.t θ_t

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta_t} J =$$

$$= \theta_t - \eta \nabla_{\theta_t} [\ell(\theta_t) + \lambda ||\theta_t||^2]$$

$$= \theta_t - \eta [\nabla_{\theta_t} \ell(\theta_t) + \lambda \nabla_{\theta_t} ||\theta_t||^2)$$

$$\boxed{= \theta_t - \eta (g_t + 2\lambda \theta_t)}$$

where $g(t) = \nabla_{\theta_t} \ell(\theta)$

[10pts] Q3.2: L1 regularization

Now let's consider L1 regularization: our objective in this case is $\mathcal{L}(\theta) + \lambda||\theta||_1$. Derive the update rule.

(Technically this becomes *Sub-Gradient* Descent since the L1 norm is not differentiable at 0. But practically it is usually not an issue.)

Problem 3.2

Now consider the CE loss function, with

$$J = \ell(\theta) + \lambda||\theta||$$

$$\theta_{t+1} = \theta_t - \mu \nabla_{\theta_t} J =$$

$$= \theta_t - \mu \cdot \left[\nabla_{\theta_t} \ell(\theta) + \lambda \nabla_{\theta_t} ||\theta||\right]$$

$$= \boxed{\theta_t - \mu g_t + \lambda}$$

where $g_t = \nabla_{\theta_t} \ell(\theta)$

In [ ]:

Problem 3.1

First we write our update rule for θ_{t+1}

$$\theta_{t+1} = \theta_t - \mu \nabla_{\theta_t} J ,$$

Now we consider the Cross-Entropy function with L-2 regularization given by

$$J = \ell(\theta) + \lambda \|\theta\|^2$$

We know tore the gradient of J w.r.t θ_t

$$\theta_{t+1} = \theta_t - \mu \nabla_{\theta_t} J =$$

$$= \theta_t - \mu \nabla_{\theta_t} [\ell(\theta_t) + \lambda \|\theta_t\|^2]$$

$$= \theta_t - \mu[\nabla_{\theta_t} \ell(\theta_t) + \lambda \nabla_{\theta_t} \|\theta_t\|^2)$$

$$\boxed{= \theta_t - \mu(g_t + 2\lambda \theta_t)}$$

where $g(t) = \nabla_{\theta_t} \ell(\theta)$

Problem 3.2

Now consider the CE loss function, with

$$J = \ell(\theta) + \lambda \|\theta\|$$

$$\theta_{t+1} = \theta_t - \mu \nabla_{\theta_t} J =$$

$$= \theta_t - \mu \cdot \left[\nabla_{\theta_t} \ell(\theta) + \lambda \nabla_{\theta_t} \|\theta\| \right]$$

$$= \boxed{\theta_t - \mu g_t + \lambda}$$

where $g_t = \nabla_{\theta_t} \ell(\theta)$