

ECE-6524 / CS-6524 Deep Learning

Assignment 1 [90 pts]

In this assignment, **you need to complete the following three sections:**

1. SVM Classifier
 - Calculating loss and gradient
2. Linear Classifier with Softmax
 - Calculating loss and gradient
3. Simple Neural Network with Multilayer Perceptrons
 - Read the forward and backward propagation
 - Experiment with simple NN

This assignment is inspired and adopted from Stanford CS231n, UIUC Deep Learning course and Deep Learning Lab from Professor Vicente Ordonez at the University of Virginia.

Submission guideline

1. Click the Save button at the top of the Jupyter Notebook.
2. Please make sure to have entered your Virginia Tech PID below.
3. Select Cell -> All Output -> Clear. This will clear all the outputs from all cells (but will keep the content of all cells).
4. Select Cell -> Run All. This will run all the cells in order.
5. Once you've rerun everything, select File -> Download as -> PDF via LaTeX
6. Look at the PDF file and make sure all your solutions are there, displayed correctly.
7. Zip BOTH the PDF file and this notebook. Remember
8. Submit your zipped file.

While you are encouraged to discuss with your peers, all work submitted is expected to be your own. If you use any information from other resources (e.g. online materials), you are required to cite it below your VT PID. Any violation will result in a 0 mark for the assignment.

Please Write Your VT PID Here: 906213559

Reference (if any):

<https://zhuanlan.zhihu.com/p/30965514> (<https://zhuanlan.zhihu.com/p/30965514>)

<https://zhuanlan.zhihu.com/p/21478575> (<https://zhuanlan.zhihu.com/p/21478575>)

<https://ljvmiranda921.github.io/notebook/2017/02/14/softmax-classifier/>
(<https://ljvmiranda921.github.io/notebook/2017/02/14/softmax-classifier/>)

<https://scikit-learn.org/stable/modules/classes.html> (<https://scikit-learn.org/stable/modules/classes.html>)

https://www.cs.virginia.edu/~vicente/recognition/notebooks/deep_learning_lab_for_vision.html
(https://www.cs.virginia.edu/~vicente/recognition/notebooks/deep_learning_lab_for_vision.html)

https://www.programcreek.com/python/example/11344/numpy.zeros_like
(https://www.programcreek.com/python/example/11344/numpy.zeros_like)

In this homework, you would need to use **Python 3.6+** along with the following packages:

1. numpy
2. matplotlib
3. scipy

All packages above can be installed through `pip install`.

In addition, you will need to have [CIFAR10](https://www.cs.toronto.edu/~kriz/cifar.html) (<https://www.cs.toronto.edu/~kriz/cifar.html>) dataset in your `cifar10/cifar-10-batches-py` folder. We provide a bash script `cifar10/get_dataset.sh` to help you download and unpack the dataset.

Once you are done with the things above, you can proceed to execute the following cells:

In [1]:

```
from google.colab import drive
drive.mount('/content/drive')
```

Drive already mounted at `/content/drive`; to attempt to forcibly remount, call `drive.mount("/content/drive", force_remount=True)`.

In [2]:

```
import sys
prefix = '/content/drive/My Drive/'
# modify "customized_path_to_your_homework", path of folder, where you uploaded your homework
customized_path_to_your_homework = 'Colab Notebooks/Assignment_1/'
sys_path = prefix + customized_path_to_your_homework
sys.path.append(sys_path)
```

In [3]:

```
import random
import numpy as np
from data_process import get_CIFAR10_data
import math
import matplotlib.pyplot as plt
%load_ext autoreload
%autoreload 2
%matplotlib inline
```

Loading CIFAR-10

In the following cells, we would first load the CIFAR10 dataset, and prepare them for our classifier:

In [4]:

```
# You can change these numbers for experimentation
# For submission we will use the default values
TRAIN_IMAGES = 49000
VAL_IMAGES = 1000
TEST_IMAGES = 1000
```

In [5]:

```
data = get_CIFAR10_data(TRAIN_IMAGES, VAL_IMAGES, TEST_IMAGES, subtract_mean=False)
X_train, y_train = data['X_train'], data['y_train']
X_val, y_val = data['X_val'], data['y_val']
X_test, y_test = data['X_test'], data['y_test']
```

In [6]:

```
print(X_train.shape)
print(y_train.shape)
print(X_val.shape)
print(y_val.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(49000, 32, 32, 3)
(49000,)
(1000, 32, 32, 3)
(1000,)
(1000, 32, 32, 3)
(1000,)
```

Now, we visualize the dataset to get a better idea of it.

In [7]:

```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```



We then subtract all the images with the **mean image** of the training set. This trick is commonly used in Computer Vision community, as it arranges the features into a range that is more manageable for the models.

In [8]:

```
mean_image = np.mean(X_train, axis=0)
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
```

Now, we convert the sets of images from dimensions of **(N, 3, 32, 32)** -> **(N, 3072)** where N is the number of images so that each **3x32x32** image is represented by a single vector. This allows us to feed the data into our classifiers.

In [9]:

```
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
```

In [10]:

```
print(X_train.shape)
print(X_val.shape)
print(X_test.shape)
```

```
(49000, 3072)
(1000, 3072)
(1000, 3072)
```

Here, we concatenate all the features with ones. This trick is particularly convenient, as in this case, we only need to deal with a single weight matrix \tilde{W} instead of both W and bias b in our training code.

In [11]:

```
X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
```

In [12]:

```
# you should get the following output:
# (49000, 3073)
# (1000, 3073)
# (1000, 3073)
print(X_train.shape)
print(X_val.shape)
print(X_test.shape)
```

```
(49000, 3073)
(1000, 3073)
(1000, 3073)
```

Calculate Classification Accuracy

Here is a simple helper function for calculation the classification accuracy:

In [13]:

```
def get_acc(pred, y_test):
    return np.sum(y_test==pred)/len(y_test)*100
```

In [14]:

```
# test get_acc
print(get_acc(y_train, y_train))
```

```
100.0
```

Section 1. Support Vector Machines (with SGD) [30 pts]

In this section, you will need to implement a **soft margin SVM** for a **multi-class classification problem**. In the soft margin SVM, you will maximize the margin between positive and negative training examples and penalize margin violations using a hinge loss.

We will optimize the SVM loss using SGD. This means you must compute the loss function with respect to model weights. You will use this gradient to update the model weights.

SVM optimized with SGD has 3 hyperparameters that you can experiment with :

- `lr` - the learning rate of your model. This parameter scales by how much the weights are changed according to the calculated gradient update
- `n_iter` - the number of training batches you are gonna use for updating your model.
- `reg_const` - Hyperparameter to determine the strength of regularization. In this case it is a coefficient on the term which maximizes the margin.

1.1. Implementing SVM with SGD [25 pts]

Recall that the multiclass SVM loss can be written as:

$$s = f(x_i, W)$$
$$L_i = \sum_{j \neq y_i} [\max(0, s_j - s_{y_i} + \text{margin})]$$

where $s = s_j$ is the vector of prediction scores of all class, s_j is the prediction score for class j , and y_i is the ground truth class label. The margin is usually set to 1. Now, to learn the weights W , we can differentiate the function L_i to get the gradient:

$$\nabla_{w_{y_i}} L_i = - \left(\sum_{j \neq y_i} 1(w_j^T x_i - w_{y_i}^T x_i + 1 > 0) \right) x_i$$

where 1 is the indicator function that is one if the condition inside is true or zero otherwise, and w_j represents the row j in your weight matrix W . To implement this, you'd simply count the number of classes that didn't meet the desired margin 1 (and hence contributed to the loss function) and then the data vector x_i scaled by this number is the gradient. Note that, in the equation above, we are only calculating the gradient with respect to the row of W that corresponds to the correct class y_i . For the other rows $j \neq y_i$, we have:

$$\nabla_{w_j} L_i = 1(w_j^T x_i - w_{y_i}^T x_i + 1 > 0) x_i$$

Now, we have derived the gradient expression above, it is time to turn these equations into the actual running code. We provide a template for implementing your SVM classifier below. You have to fill in the **TODO** part, and train your classifier.

In [15]:

```
class SVM():
    def __init__(self, input_size, output_size):
        """
        Initialises Softmax classifier with initializing
        weights, alpha(learning rate), number of epochs
        and regularization constant.
        """
        self.input_size = input_size
        self.output_size = output_size
        self.w = np.random.randn(input_size, output_size) * 0.001

    def calc_grad_and_loss(self, X_train, y_train, reg, margin):
        """
        Calculate gradient of the svm hinge loss.

        Inputs have dimension D, there are C classes, and we operate on minibatches
        of N examples.

        Inputs:
        - X_train: A numpy array of shape (N, D) containing a minibatch of data.
        - y_train: A numpy array of shape (N,) containing training labels; y[i] = c means
        that X[i] has label c, where 0 <= c < C.
        - reg: A float number Regularization strength;
        - margin: A float number margin for the multi-class SVM loss (usually set to
        1);

        Returns:
        - gradient with respect to weights w; an array of same shape as w;
        - multi-class SVM loss;
        """
        #####
        #
        # TODO:
        #
        # Calculate the loss and gradient for the SVM classifier.
        #
        #####
        #
        n_train = len(X_train) # number of training samples
        scores = X_train @ self.w
        scores_correct = scores[np.arange(n_train), y_train]

        # computing losses for each classes
        losses = scores - scores_correct[:, np.newaxis] + margin
        losses = (losses > 0) * losses
        losses[np.arange(n_train), y_train] = 0

        loss = np.mean(losses) + 0.5 * reg * np.sum(np.square(self.w)) # calculating overall loss

        # calculating gradient
        W = (losses > 0).astype('float')
        row_sum = np.sum(W, axis=1)
        W[np.arange(n_train), y_train] = -row_sum
        grad_w = (X_train.T @ W) / n_train + reg * self.w
        #####
        #
```

```

#                                     END OF YOUR CODE
#
#####
#

    return grad_w, loss

def train(self, X_train, y_train, lr=0.01, reg_const=0.05, margin=1.0,
          n_iters=1500, batch_size=200):
    """
    Train SVM classifier by taking one Stochastic Gradient Descent update
    on the input minibatch.

    Inputs:
    - X_train: A numpy array of shape (N, D) containing training data;
    N examples with D dimensions
    - y_train: A numpy array of shape (N,) containing training labels;
    - lr: A float number Learning rate;
    - reg_const: A float number Regularization strength;
    - margin: A float number margin for the multi-class SVM Loss (usually set to
1);
    - n_iters: An integer number indicating the number of training iteration;
    - batch_size: An integer indicating the number of samples per batch;
    Returns:
    - loss: Loss for this particular input batch.
    """
    loss_hist = []

    for i in range(n_iter):
        batch_indices = np.random.choice(len(X_train), batch_size)
        X_batch = X_train[batch_indices]
        y_batch = y_train[batch_indices]
        grad, loss = self.calc_grad_and_loss(X_batch, y_batch, reg_const, margin)
        self.w -= lr * grad
        if (i + 1) % 100 == 0:
            print(f'Iter {i+1}/{n_iter} - loss: {loss} ')
        loss_hist.append(loss)
    return loss_hist

def predict(self, X_test):
    """
    Use the trained weights of svm classifier to predict labels for
    data points.

    Inputs:
    - X_test: A numpy array of shape (N, D) containing training data; there are N
    training samples each of dimension D.

    Returns:
    - pred: Predicted labels for the data in X_test. pred is a 1-dimensional
    array of length N, and each element is an integer giving the predicted
    class.
    """
    #####
#
# TODO:
#
# Make a prediction based on your classifier weights
#
#####
#

```



```

pred_test = X_test @ self.w
pred = np.argmax(pred_test, axis=1)
#####
#
#                                     END OF YOUR CODE
#
#####
#
return pred

```

Train your SVM

Now, train your SVM classifier. You should expect the loss decreases as the training goes on.

In [16]:

```

n_iter = 1500
batch_size = 200
num_classes = np.max(y_train) + 1
num_features = X_train.shape[1]
svm = SVM(num_features, num_classes)
loss_hist = svm.train(X_train, y_train, lr=1e-7, reg_const=5e3, n_iters=1500,)

```

```

Iter 100/1500 - loss: 70.24199776667258
Iter 200/1500 - loss: 63.139001055398495
Iter 300/1500 - loss: 56.90547526059544
Iter 400/1500 - loss: 51.419357633761884
Iter 500/1500 - loss: 46.33052848561495
Iter 600/1500 - loss: 41.958924142158764
Iter 700/1500 - loss: 37.83464933341287
Iter 800/1500 - loss: 34.13435102714803
Iter 900/1500 - loss: 30.81729015286214
Iter 1000/1500 - loss: 27.8822980494208
Iter 1100/1500 - loss: 25.159923050892957
Iter 1200/1500 - loss: 22.796307030353066
Iter 1300/1500 - loss: 20.57999325760777
Iter 1400/1500 - loss: 18.66967622755152
Iter 1500/1500 - loss: 16.919026610076962

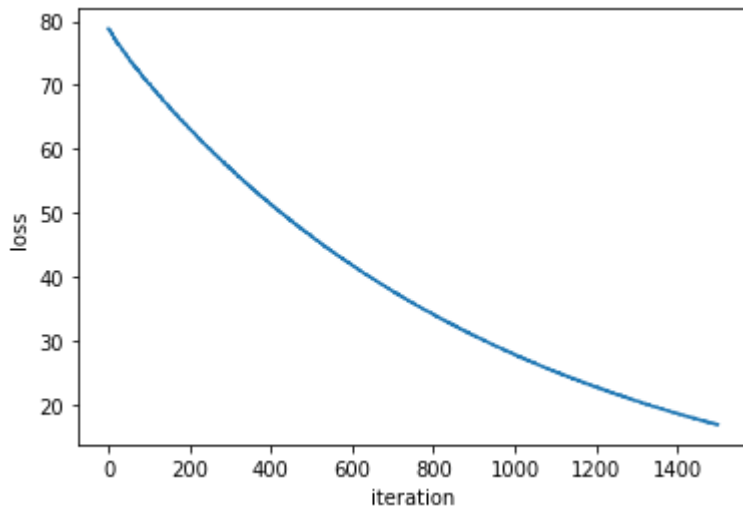
```

Plot the Loss Curve

Plotting the loss curve is a proper etiquette for Machine Learning/Deep Learning practitioner.

In [17]:

```
plt.plot(loss_hist);  
plt.ylabel('loss');  
plt.xlabel('iteration');
```



Accuracy?

If the SVM is implemented correctly, you should get an training/validation accuracy around 34%-35%.

In [18]:

```
pred_svm = svm.predict(X_train)  
print('The training accuracy is given by : %f' % (get_acc(pred_svm, y_train)))  
pred_svm = svm.predict(X_val)  
print('The validation accuracy is given by : %f' % (get_acc(pred_svm, y_val)))
```

The training accuracy is given by : 34.304082

The validation accuracy is given by : 36.000000

Section 1.2. Improve your SVM [5 pts]

Now, try to improve the accuracy of your classifier. You can perform hyperparameter search with cross-validation, or conduct some extra preprocessing on the data. **Note that you should never use the `X_test` and `y_test` for tuning your model.**

In [28]:

```
# How can you improve it
# cross validation
class CrossValidation(object):
    def __init__(self, X, y):
        self.X = X
        self.y = y

    def validate(self, models, **kwargs):
        num_folds = len(models)

        # split data
        X_train_folds = np.array_split(self.X, num_folds)
        y_train_folds = np.array_split(self.y, num_folds)

        # validate
        accuracy = np.zeros(num_folds)
        for i in range(num_folds):
            X_val = X_train_folds[i]
            X_train = np.concatenate([X_train_folds[j] for j in range(num_folds) if j != i], axis=0)
            y_val = y_train_folds[i]
            y_train = np.concatenate([y_train_folds[j] for j in range(num_folds) if j != i], axis=0)
            model = models[i]
            model.train(X_train, y_train, **kwargs)
            pred = model.predict(X_val)
            accuracy[i] = get_acc(pred, y_val)
            print('Fold %d, accuracy = %f' % (i, accuracy[i]))
        return np.mean(accuracy)

import os
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter

# parameters
learning_rates = 4 ** np.arange(3) * 1e-8
regularization_strengths = 4 ** np.arange(3) * 1e2

if os.path.exists('svm_cross_val.txt'):
    lr_list, reg_list, accuracy_list = [], [], []
    f = open('svm_cross_val.txt', 'r')
    for line in f.readlines():
        lr, reg, accuracy = map(float, line.strip().split(','))
        accuracy_list.append(accuracy)
    f.close()
    for lr in learning_rates:
        for reg in regularization_strengths:
            lr_list.append(lr)
            reg_list.append(reg)

    lr = np.array(lr_list)
    reg = np.array(reg_list)
    accuracy = np.array(accuracy_list)

# print best hyper-parameters
best_hyper_parameters = np.argmax(accuracy)
print('best learning rate = %s' % str(lr[best_hyper_parameters]))
```

```
print('best regularization = %s' % str(reg[best_hyper_parameters]))
print('validation accuracy = %s%%' % str(accuracy[best_hyper_parameters]))
```

else:

```
n_iter = 1500
batch_size = 200
folds = 5
num_classes = np.max(y_train) + 1
num_features = X_train.shape[1]

f = open('svm_cross_val.txt', 'w')
cross_validation = CrossValidation(X_train, y_train)
for lr in learning_rates:
    for reg in regularization_strengths:
        # the models
        svms = [SVM(num_features, num_classes) for _ in range(folds)]
        accuracy = cross_validation.validate(svms, lr=lr, reg_const=reg, n_iters=1500)

        print(lr, reg, accuracy)
        f.write('%s,%s,%s\n' % (str(lr), str(reg), str(accuracy)))
f.close()
```

```
best learning rate = 1.6e-07
best regularization = 6400.0
validation accuracy = 37.34897959183673%
```

In [20]:

```
#best parameters
n_iter = 1500
batch_size = 200
num_classes = np.max(y_train) + 1
num_features = X_train.shape[1]
svm = SVM(num_features, num_classes)
loss_hist = svm.train(X_train, y_train, lr=1.6e-07, reg_const=6400, n_iters=1500)

pred_svm = svm.predict(X_train)
print('The training accuracy is: %f' % (get_acc(pred_svm, y_train)))
pred_svm = svm.predict(X_val)
print('The validation accuracy is: %f' % (get_acc(pred_svm, y_val)))
```

```
Iter 100/1500 - loss: 80.57850738170536
Iter 200/1500 - loss: 65.34679402467204
Iter 300/1500 - loss: 53.043059296525556
Iter 400/1500 - loss: 43.03501708272114
Iter 500/1500 - loss: 34.910343948632594
Iter 600/1500 - loss: 28.41683599369182
Iter 700/1500 - loss: 23.208706462691275
Iter 800/1500 - loss: 18.915550790236065
Iter 900/1500 - loss: 15.392990348961886
Iter 1000/1500 - loss: 12.645288398263062
Iter 1100/1500 - loss: 10.401012139729147
Iter 1200/1500 - loss: 8.545877052323233
Iter 1300/1500 - loss: 7.07261302226506
Iter 1400/1500 - loss: 5.842565205663444
Iter 1500/1500 - loss: 4.8033167212013375
The training accuracy is: 38.436735
The validation accuracy is: 39.400000
```

Test SVM

Evaluate the testing accuracy of your SVM:

In [21]:

```
pred_svm = svm.predict(X_test)
print('The testing accuracy is given by : %f' % (get_acc(pred_svm, y_test)))
```

The testing accuracy is given by : 38.200000

Visualization

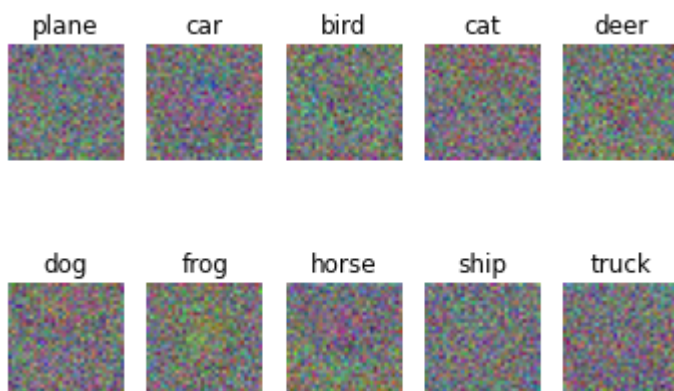
Visualize the weights of your best model.

In [22]:

```
# Visualize the learned weights for each class.
# Hint: reshape your weights (w/o bias) to (32, 32, 3, 10), then use min-max normalization and rescale them back to 0-255.
w = svm.w[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)
w_min, w_max = np.min(w), np.max(w)

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

# plot for each class
for i in range(10):
    plt.subplot(2, 5, i + 1)
    # Rescale the weights to be between 0 and 255
    w_255 = 255.0 * (w[:, :, :, i].squeeze() - w_min) / (w_max - w_min)
    plt.imshow(w_255.astype('uint8'))
    plt.axis('off')
    plt.title(classes[i])
```



Describe what your visualized SVM weights look like, and offer a brief explanation for why they look the way that they do.

Your comments: Each figure representing a template of a class, and this visualization are similar to a mean map of that class of images.

Section 2. Softmax Classifier (with SGD) [30 pts]

Next, you will train a Softmax classifier. This classifier consists of a linear function of the input data followed by a softmax function which outputs a vector of dimension C (number of classes) for each data point. Each entry of the softmax output vector corresponds to a confidence in one of the C classes, and like a probability distribution, the entries of the output vector sum to 1.

Section 2.1. Implement the Softmax Classifier [25 pts]

Recall that, from the lecture, we can write down the loss function as:

$$s = f(x_i, W)$$
$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

And, through some calculations, we can obtain the expression for the gradient:

$$\nabla_{w_l} L_i = \left(\frac{e^{w_l^T x_i}}{\sum_j e^{w_j^T x_i}} - 1(y_i = l) \right) x_i$$

The softmax classifier also has 3 hyperparameters that you can experiment with :

- `lr` - the learning rate of your model. This parameter scales by how much the weights are changed according to the calculated gradient update
- `n_iter` - the number of training batches you are gonna use for updating your model.
- `reg_const` - Hyperparameter to determine the strength of regularization. In this case it is a coefficient on the term which maximizes the margin.

Same as the previous section, we provide a template for you to implement your classifier:

In [23]:

```

class Softmax_Classfier():
    def __init__(self, input_size, output_size):
        """
        Initialises Softmax classifier with initializing
        weights, alpha(learning rate), number of epochs
        and regularization constant.
        """
        self.input_size = input_size
        self.output_size = output_size
        self.w = np.random.randn(input_size, output_size) * 0.001

    def softmax(self, logits):
        #####
        # TODO:
        # Implement the softmax function here.
        #####
        probs = np.exp(logits) / np.sum(np.exp(logits), axis=1)[:, np.newaxis]
        #####
        #
        # END OF YOUR CODE
        #####
        return probs

    def calc_grad_and_loss(self, X_train, y_train, reg_const):
        """
        Calculate gradient of the svm hinge loss.

        Inputs have dimension D, there are C classes, and we operate on minibatches
        of N examples.

        Inputs:
        - X_train: A numpy array of shape (N, D) containing a minibatch of data.
        - y_train: A numpy array of shape (N,) containing training labels; y[i] = c means
            that X[i] has label c, where 0 <= c < C.
        - reg: A float number Regularization strength;

        Returns:
        - gradient with respect to weights w; an array of same shape as w;
        - multi-class SVM loss;
        """
        #####
        # TODO:
        # Calculate the loss and gradient for the SVM classifier.
        #####
        num_train = X_train.shape[0]
        logits = X_train @ self.w
        prob = self.softmax(logits)
        loss = - np.log(prob[np.arange(num_train), y_train])

```

```

loss = np.mean(loss)
prob[ $\text{np.arange}(\text{num\_train})$ , y_train] -= 1
grad_w = X_train.T @ prob / num_train + reg_const * self.w
#####

#
#                                     END OF YOUR CODE
#####

return grad_w, loss

def train(self, X_train, y_train, lr=0.01, reg_const=0.05,
          n_iters=1500, batch_size=200):
    """
    Train softmax classifier by taking one Stochastic Gradient Descent update
    on the input minibatch.

    Inputs:
    - X_train: A numpy array of shape (N, D) containing training data;
      N examples with D dimensions
    - y_train: A numpy array of shape (N,) containing training labels;
    - lr: A float number learning rate;
    - reg_const: A float number Regularization strength;
    - n_iters: An integer number indicating the number of training iteration;
    - batch_size: An integer indicating the number of samples per batch;
    Returns:
    - loss: Loss for this particular input batch.
    """
    loss_hist = []

    for i in range(n_iter):
        batch_indices = np.random.choice(len(X_train), batch_size)
        X_batch = X_train[batch_indices]
        y_batch = y_train[batch_indices]
        grad, loss = self.calc_grad_and_loss(X_batch, y_batch, reg_const)
        self.w -= lr * grad
        if (i + 1) % 100 == 0:
            print(f'Iter {i+1}/{n_iter} - loss: {loss}')
        loss_hist.append(loss)
    return loss_hist

def predict(self, X_test):
    """
    Use the trained weights of svm classifier to predict labels for
    data points.

    Inputs:
    - X_test: A numpy array of shape (N, D) containing training data; there are N
      training samples each of dimension D.

    Returns:
    - pred: Predicted labels for the data in X_test. pred is a 1-dimensional
      array of length N, and each element is an integer giving the predicted
      class.
    """
    #####

#
# TODO:
#
# Make a prediction based on your classifier weights
#

```



```
#####
#
logits = X_test @ self.w
prob = self.softmax(logits)
pred = np.argmax(- np.log(prob), axis=1)
#####
#
#                                     END OF YOUR CODE
#
#####
#
return pred
```

In [24]:

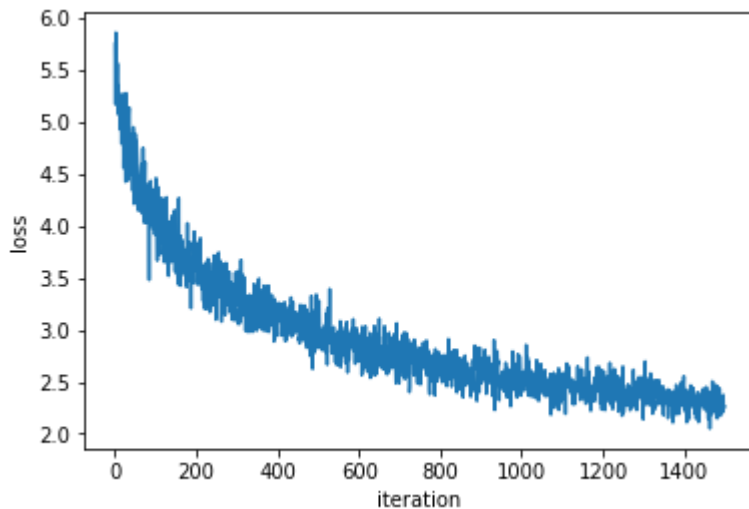
```
n_iter = 1500
batch_size = 200
num_classes = np.max(y_train) + 1
num_features = X_train.shape[1]
softmax_classifier = Softmax_Classfier(num_features, num_classes)
loss_hist = softmax_classifier.train(X_train, y_train, lr=1e-7, reg_const=2e3, n_iters=
1500, batch_size=batch_size)
```

```
Iter 100/1500 - loss: 4.014992051257605
Iter 200/1500 - loss: 3.578513778657579
Iter 300/1500 - loss: 3.550826883154191
Iter 400/1500 - loss: 2.9655756656168637
Iter 500/1500 - loss: 2.9899417039695724
Iter 600/1500 - loss: 2.696713824502743
Iter 700/1500 - loss: 2.4864243227464566
Iter 800/1500 - loss: 2.565024957392536
Iter 900/1500 - loss: 2.5056405752538513
Iter 1000/1500 - loss: 2.474907348584396
Iter 1100/1500 - loss: 2.387673811673159
Iter 1200/1500 - loss: 2.5003871276837866
Iter 1300/1500 - loss: 2.1394415781329768
Iter 1400/1500 - loss: 2.5536852168818194
Iter 1500/1500 - loss: 2.2638559159147245
```

Again, plot the loss:

In [25]:

```
plt.plot(loss_hist);  
plt.ylabel('loss');  
plt.xlabel('iteration');
```



Accuracy?

If the Softmax Classifier is implemented correctly, you should get an accuracy around 29%.

In [26]:

```
pred_softmax = softmax_classifier.predict(X_train)  
print('The training accuracy is given by : %f' % (get_acc(pred_softmax, y_train)))  
pred_softmax = softmax_classifier.predict(X_val)  
print('The validation accuracy is given by : %f' % (get_acc(pred_softmax, y_val)))
```

The training accuracy is given by : 25.683673

The validation accuracy is given by : 26.400000

Section 2.1. Improve your Softmax Classifier [5 pts]

Again, try to improve the accuracy of your classifier. You can perform hyperparameter search with cross-validation, or conduct some extra preprocessing on the data. **Note that you should never use the `X_test` and `y_test` for tuning your model.** In addition, ****We expect to see an accuracy around 34% after the improvement.****

In [31]:

```
# Get the accuracy to above 34%!
# For hyperparameter search, you can try different combinations of (reg_const, margin,
  Lr)
# cross validation
class CrossValidation(object):
    def __init__(self, X, y):
        self.X = X
        self.y = y

    def validate(self, models, **kwargs):
        num_folds = len(models)

        # split data
        X_train_folds = np.array_split(self.X, num_folds)
        y_train_folds = np.array_split(self.y, num_folds)

        # validate
        accuracy = np.zeros(num_folds)
        for i in range(num_folds):
            X_val = X_train_folds[i]
            X_train = np.concatenate([X_train_folds[j] for j in range(num_folds) if j !=
i], axis=0)
            y_val = y_train_folds [i]
            y_train = np.concatenate([y_train_folds [j] for j in range(num_folds) if j !=
=i], axis=0)
            model = models[i]
            model.train(X_train, y_train, **kwargs)
            pred = model.predict(X_val)
            accuracy[i] = get_acc(pred, y_val)
            print('Fold %d, accuracy = %f' % (i, accuracy[i]))
        return np.mean(accuracy)

import os
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter

# parameters
learning_rates = 4 ** np.arange(4) * 1e-8
regularization_strengths = 4 ** np.arange(4) * 1e2

if os.path.exists('softmax_cross_val.txt'):
    lr_list, reg_list, accuracy_list = [], [], []
    f = open('softmax_cross_val.txt', 'r')
    for line in f.readlines():
        lr, reg, accuracy = map(float, line.strip().split(','))
        accuracy_list.append(accuracy)
    f.close()
    for lr in learning_rates:
        for reg in regularization_strengths:
            lr_list.append(lr)
            reg_list.append(reg)

    lr = np.array(lr_list)
    reg = np.array(reg_list)
    accuracy = np.array(accuracy_list)
```

```

# print best hyper-parameters
best_hyper_parameters = np.argmax(accuracy)
print('best learning rate = %s' % str(lr[best_hyper_parameters]))
print('best regularization = %s' % str(reg[best_hyper_parameters]))
print('validation accuracy = %s%%' % str(accuracy[best_hyper_parameters]))

else:
    n_iter = 1500
    batch_size = 200
    folds = 5
    num_classes = np.max(y_train) + 1
    num_features = X_train.shape[1]

    f = open('softmax_cross_val.txt', 'w')
    cross_validation = CrossValidation(X_train, y_train)
    for lr in learning_rates:
        for reg in regularization_strengths:
            # the models
            softmaxs = [Softmax_Classfier(num_features, num_classes) for _ in range(folds)]
            accuracy = cross_validation.validate(softmaxs, lr=lr, reg_const=reg, n_iters=1500)
            print(lr, reg, accuracy)
            f.write('%s,%s,%s\n' % (str(lr), str(reg), str(accuracy)))
    f.close()

```

```

best learning rate = 6.4e-07
best regularization = 1600.0
validation accuracy = 38.33877551020409%

```

In [37]:

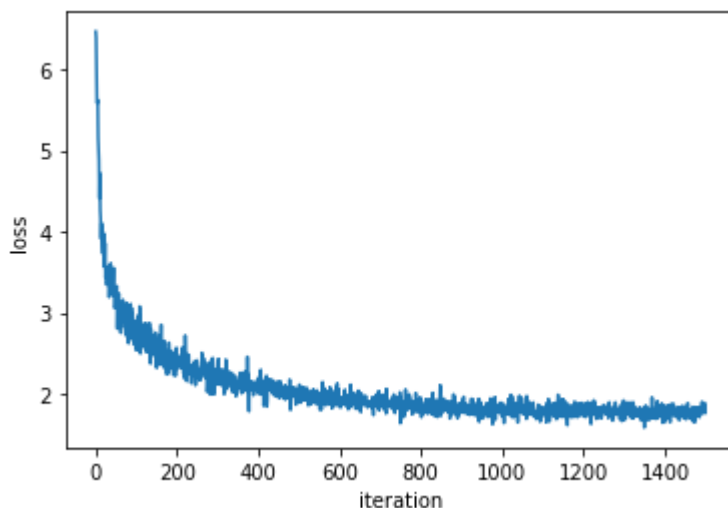
```
# use best parameters
n_iter = 1500
batch_size = 200
num_classes = np.max(y_train) + 1
num_features = X_train.shape[1]
softmaxs = Softmax_Classfier(num_features, num_classes)
loss_hist = softmaxs.train(X_train, y_train, lr=6.4e-07, reg_const=1600.0, n_iters=1500,
,)

plt.plot(loss_hist)
plt.ylabel('loss')
plt.xlabel('iteration')
```

```
Iter 100/1500 - loss: 2.905392330970659
Iter 200/1500 - loss: 2.5690084738817713
Iter 300/1500 - loss: 2.324564854485022
Iter 400/1500 - loss: 2.121803718577397
Iter 500/1500 - loss: 1.9589776631411746
Iter 600/1500 - loss: 1.8991090465910405
Iter 700/1500 - loss: 2.0014762086540516
Iter 800/1500 - loss: 1.8494899958935678
Iter 900/1500 - loss: 1.847843450578158
Iter 1000/1500 - loss: 1.7321598016761155
Iter 1100/1500 - loss: 1.8145105452421968
Iter 1200/1500 - loss: 1.7588636296151525
Iter 1300/1500 - loss: 1.6846473100866997
Iter 1400/1500 - loss: 1.7023998788242622
Iter 1500/1500 - loss: 1.8144963736064335
```

Out[37]:

Text(0.5, 0, 'iteration')



Testing Softmax

Finally, evaluate the testing accuracy.

In [39]:

```
pred_softmax = softmaxs.predict(X_test)
print('The testing accuracy is given by : %f' % (get_acc(pred_softmax, y_test)))
```

The testing accuracy is given by : 36.000000

Visualization

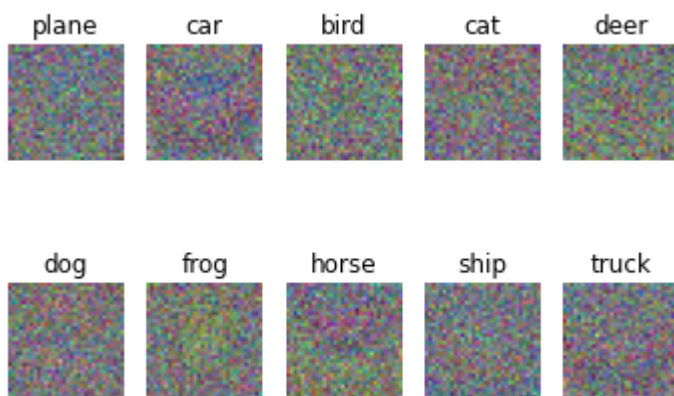
Visualize the weights of your best model.

In [40]:

```
# Visualize the learned weights for each class.
# Hint: reshape your weights (no bias) to (32, 32, 3, 10), and then rescale them back to 0-255 using min-max normalization.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
# plot for each class
w = softmaxs.w[:-1, :] # strip out the bias
w = w.reshape(32, 32, 3, 10)
w_min, w_max = np.min(w), np.max(w)

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

# plot for each class
for i in range(10):
    plt.subplot(2, 5, i + 1)
    # Rescale the weights to be between 0 and 255
    w_255 = 255.0 * (w[:, :, :, i].squeeze() - w_min) / (w_max - w_min)
    plt.imshow(w_255.astype('uint8'))
    plt.axis('off')
    plt.title(classes[i])
```



Describe what your visualized weights look like compared to SVM weights.

Your comments: Compared with SVM weight visualization, it feels like you can see a little outline of the classes.

Section 3. Multilayer Perceptron (MLP)

In this section, we present you a modified version of simple neural network from Professor Vicente Ordonez at the University of Virginia. Your job is to read through the implementation (optional, but highly recommended), and try the MLP classifier. For additional resources on understanding forward and backpropagation, you can check the materials here: <http://cs231n.github.io/optimization-2/> (<http://cs231n.github.io/optimization-2/>)

Consider a single-layer supervised neural network that has 4 inputs and 3 outputs. First, let's review the skeleton of a single linear layer neural network. The inputs of the network are the variables x_1, x_2, x_3, x_4 , or the input vector $\mathbf{x} = [x_1, x_2, x_3, x_4]$, the outputs of the network are $\hat{y}_1, \hat{y}_2, \hat{y}_3$, or the output vector $\hat{\mathbf{y}} = [\hat{y}_1, \hat{y}_2, \hat{y}_3]$:



The given j -th output \hat{y}_j of this single linear layer + activation function is computed as follows:

$$\hat{y}_j = \text{sigmoid}(w_{1j}x_1 + w_{2j}x_2 + w_{3j}x_3 + w_{4j}x_4 + b_j) = \text{sigmoid}\left(\sum_{i=1}^{i=4} w_{ij}x_i + b_j\right)$$

In matrix notation, this would be:

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \end{bmatrix}^T = \text{Sigmoid}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}^T \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,1} & w_{3,2} & w_{3,3} \\ w_{4,1} & w_{4,2} & w_{4,3} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}^T\right)$$

or more compactly:

$$\hat{\mathbf{y}}^T = \text{Sigmoid}(\mathbf{x}^T \cdot \mathbf{W} + \mathbf{b}^T)$$

The element-wise sigmoid function is: $\text{Sigmoid}(\mathbf{x}) = 1 / (1 + \exp(-\mathbf{x}))$, or alternatively:

$\text{Sigmoid}(\mathbf{x}) = \exp(\mathbf{x}) / (1 + \exp(\mathbf{x}))$. Here the sigmoid is separated logically into an activation layer $\sigma(x)$ and a linear layer `linear(3, 4)` as illustrated in figure.

Training these weights \mathbf{W} and biases \mathbf{b} requires having many training pairs $(\hat{\mathbf{y}}^{(m)}, \mathbf{x}^{(m)})$. The inputs \mathbf{x} can be the pixels of an image, indices of words, the entries in a database, and the outputs $\hat{\mathbf{y}}$ can also be literally anything, including a number indicating a category, a set of numbers indicating the indices of words composing a sentence, an output image itself, etc.

3.1. Forward-propagation

Computing the outputs $\hat{\mathbf{y}}$ from the inputs \mathbf{x} in this network composed of a single linear layer, and a sigmoid layer, is called forward-propagation. Below is the code that implements these two operations:

In [41]:

```
import numpy as np
import numpy.matlib

class nn_Sigmoid:
    def forward(self, x):
        return 1 / (1 + np.exp(-x))

class nn_Linear:
    def __init__(self, input_dim, output_dim):
        # Initialized with random numbers from a gaussian N(0, 0.001)
        self.weight = np.matlib.randn(input_dim, output_dim) * 0.001
        self.bias = np.matlib.randn((1, output_dim)) * 0.001

        # y = Wx + b
    def forward(self, x):
        return np.dot(x, self.weight) + self.bias

    def getParameters(self):
        return [self.weight, self.bias]

# Let's test the composition of the two functions (forward-propagation in the neural network).
x1 = np.array([[1, 2, 2, 3]])
y_hat1 = nn_Sigmoid().forward(nn_Linear(4, 3).forward(x1))
print('x[1] = ' + str(x1))
print('y_hat[1] = ' + str(y_hat1) + '\n')

# Let's test the composition of the two functions (forward-propagation in the neural network).
x2 = np.array([[4, 5, 2, 1]])
y_hat2 = nn_Sigmoid().forward(nn_Linear(4, 3).forward(x2))
print('x[2] = ' + str(x2))
print('y_hat[2] = ' + str(y_hat2) + '\n')

# We can also compute both at once, which could be more efficient since it requires a single matrix multiplication.
x = np.concatenate((x1, x2), axis = 0)
y_hat = nn_Sigmoid().forward(nn_Linear(4, 3).forward(x))
print('x = ' + str(x))
print('y_hat = ' + str(y_hat))
```

```
x[1] = [[1 2 2 3]]
y_hat[1] = [[0.50113142 0.50074325 0.49937981]]
```

```
x[2] = [[4 5 2 1]]
y_hat[2] = [[0.50029084 0.49885051 0.49959177]]
```

```
x = [[1 2 2 3]
      [4 5 2 1]]
y_hat = [[0.49999333 0.49979075 0.49643297]
          [0.49937219 0.500278    0.49440702]]
```


3.2. Loss functions.

After computing the output predictions $\hat{\mathbf{y}}$ it is necessary to compare these against the true values of \mathbf{y} . Let's call these true, correct, or desired values \mathbf{y} . Typically, a simple loss or cost function is used to measure the degree by which the prediction $\hat{\mathbf{y}}$ is wrong with respect to \mathbf{y} . A common loss function for regression is the sum of squared differences between the prediction and its true value. Assuming a prediction $\hat{\mathbf{y}}^{(d)}$ for our training sample $\mathbf{x}^{(d)}$ with true value $\mathbf{y}^{(d)}$, then the loss can be computed as:

$$loss(\hat{\mathbf{y}}^{(d)}, \mathbf{y}^{(d)}) = (\hat{y}_1^{(d)} - y_1^{(d)})^2 + (\hat{y}_2^{(d)} - y_2^{(d)})^2 + (\hat{y}_3^{(d)} - y_3^{(d)})^2 = \sum_{j=1}^3 (\hat{y}_j^{(d)} - y_j^{(d)})^2$$

The goal is to modify the parameters $[\mathbf{W}, \mathbf{b}]$ in the Linear layer so that the value of $loss(\hat{\mathbf{y}}^{(d)}, \mathbf{y}^{(d)})$ becomes as small as possible for all training samples in a set $D = \{(\mathbf{x}^{(d)}, \mathbf{y}^{(d)})\}$. This would in turn ensure that predictions $\hat{\mathbf{y}}$ are as similar as possible to the true values \mathbf{y} . To achieve this we need to minimize the following function:

$$\mathcal{L}(\mathbf{W}, \mathbf{b}) = \sum_{d=1}^{d=|D|} loss(\hat{\mathbf{y}}^{(d)}, \mathbf{y}^{(d)})$$

The only two variables for our model in the function $\mathcal{L}(\mathbf{W}, \mathbf{b})$ are \mathbf{W} and \mathbf{b} , this is because the training dataset D is fixed. Finding the values of \mathbf{W} and \mathbf{b} that minimize the the loss, particularly for complex functions, is typically done using gradient based optimization, like Stochastic Gradient Descent (SGD).

In [42]:

```
class nn_MSECriterion: # MSE = mean squared error.
    def forward(self, predictions, labels):
        return np.sum(np.square(predictions - labels))

# Let's test the loss function.
y_hat = np.array([[0.23, 0.25, 0.33], [0.23, 0.25, 0.33], [0.23, 0.25, 0.33], [0.23, 0.25, 0.33]])
y_true = np.array([[0.25, 0.25, 0.25], [0.33, 0.33, 0.33], [0.77, 0.77, 0.77], [0.80, 0.80, 0.80]])

nn_MSECriterion().forward(y_hat, y_true)
```

Out[42]:

1.6271000000000002

3.3. Backward-propagation (Backpropagation)

As we discussed in class, backpropagation is just applying the chain-rule in calculus to compute the derivative of a function which is the composition of many functions (this is essentially definition of the neural network).

Below is the implementation of required derivative computations for our simple network. You are highly advised to derive the derivatives implemented here to make sure you understand how one arrives at them.

In [43]:

```
# This is referred above as f(u).

class nn_MSECriterion:
    def forward(self, predictions, labels):
        return np.sum(np.square(predictions - labels))

    def backward(self, predictions, labels):
        num_samples = labels.shape[0]
        return 2 * (predictions - labels)

# This is referred above as g(v).
class nn_Sigmoid:
    def forward(self, x):
        return 1 / (1 + np.exp(-x))

    def backward(self, x, gradOutput):
        # It is usually a good idea to use gv from the forward pass and not recompute it again here.
        gv = 1 / (1 + np.exp(-x))
        return np.multiply(np.multiply(gv, (1 - gv)), gradOutput)

# This is referred above as h(W, b)
class nn_Linear:
    def __init__(self, input_dim, output_dim):
        # Initialized with random numbers from a gaussian  $N(0, 0.001)$ 
        self.weight = np.matlib.randn(input_dim, output_dim) * 0.001
        self.bias = np.matlib.randn((1, output_dim)) * 0.001
        self.gradWeight = np.zeros_like(self.weight)
        self.gradBias = np.zeros_like(self.bias)

    def forward(self, x):
        return np.dot(x, self.weight) + self.bias

    def backward(self, x, gradOutput):
        #  $dL/dw = dh/dw * dL/dv$ 
        self.gradWeight = np.dot(x.T, gradOutput)
        #  $dL/db = dh/db * dL/dv$ 
        self.gradBias = np.copy(gradOutput)
        self.gradBias = np.copy(np.sum(gradOutput, axis=0))
        # return  $dL/dx = dh/dx * dL/dv$ 
        return np.dot(gradOutput, self.weight.T)

    def getParameters(self):
        params = [self.weight, self.bias]
        gradParams = [self.gradWeight, self.gradBias]
        return params, gradParams

# Let's test some dummy inputs for a full pass of forward and backward propagation.
x1 = np.array([[1, 2, 2, 3]])
y1 = np.array([[0.25, 0.25, 0.25]])

# Define the operations.
linear = nn_Linear(4, 3) # h(W, b)
sigmoid = nn_Sigmoid() # g(v)
loss = nn_MSECriterion() # f(u)

# Forward-propagation.
lin = linear.forward(x1)
y_hat = sigmoid.forward(lin)
```

```

loss_val = loss.forward(y_hat, y1) # Loss function.

# Backward-propagation.
dy_hat = loss.backward(y_hat, y1)
dlin = sigmoid.backward(lin, dy_hat)
dx1 = linear.backward(x1, dlin)

print('\n num_samples = ' + str(y1.shape[0]))

# Show parameters of the Linear Layer.
print('\nW = ' + str(linear.weight))
print('\nB = ' + str(linear.bias))

# Show the intermediate outputs in the forward pass.
print('\nx1    = ' + str(x1))
print('\nlin    = ' + str(lin))
print('\ny_hat = ' + str(y_hat))

print('\nloss = ' + str(loss_val))

# Show the intermediate gradients with respect to inputs in the backward pass.
print('\ndy_hat = ' + str(dy_hat))
print('\ndlin   = ' + str(dlin))
print('\ndx1    = ' + str(dx1))

# Show the gradients with respect to parameters.
print('\ndW = ' + str(linear.gradWeight))
print('\ndB = ' + str(linear.gradBias))

```

```

num_samples = 1

W = [[-0.00036182  0.00021045  0.00050575]
      [-0.00233735 -0.00241531  0.00059431]
      [ 0.00239179 -0.00305687  0.00026237]
      [-0.00114638  0.00073156 -0.00147838]]
B = [[-3.29653394e-04 -7.81286769e-04 -4.27397727e-05]]

x1    = [[1 2 2 3]]
lin    = [[-0.00402174 -0.00932053 -0.00225879]]
y_hat = [[0.49899457 0.49766988 0.4994353 ]]

loss = 0.18555663653651155

dy_hat = [[0.49798913 0.49533977 0.49887061]]
dlin    = [[0.12449678 0.12383225 0.12471749]]
dx1     = [[ 4.40908622e-05 -5.15965716e-04 -4.80470119e-05 -2.36510556e-04]]

dW = [[0.12449678 0.12383225 0.12471749]
       [0.24899356 0.24766451 0.24943499]
       [0.24899356 0.24766451 0.24943499]
       [0.37349034 0.37149676 0.37415248]]
dB = [[0.12449678 0.12383225 0.12471749]]

```

3.4. Gradient checking

The gradients can also be computed with numerical approximation using the definition of derivatives. Let a single input pair (\mathbf{x}, \mathbf{y}) be the input, for each entry w_{ij} in the weight matrix \mathbf{W} , the partial derivatives can be computed as follows:

$$\frac{\partial \mathcal{L}(\mathbf{W}, \mathbf{b})}{\partial w_{ij}} = \frac{\mathcal{L}(\mathbf{W} + \mathbf{E}_{ij}, b) - \mathcal{L}(\mathbf{W} - \mathbf{E}_{ij}, b)}{2\epsilon},$$

where \mathbf{E}_{ij} is a matrix that has ϵ in its (i, j) entry and zeros everywhere else. Intuitively this gradient tells how would the value of the loss changes if we change a particular weight w_{ij} by an ϵ amount. We can do the same to compute derivatives with respect to the bias parameters b_i . Below is the code that checks for a given input (\mathbf{x}, \mathbf{y}) , the gradients for the matrix \mathbf{W} .

In [44]:

```
# We will compute derivatives with respect to a single data pair (x,y)
x = np.array([[2.34, 3.8, 34.44, 5.33]])
y = np.array([[3.2, 4.2, 5.3]])

# Define the operations.
linear = nn_Linear(4, 3)
sigmoid = nn_Sigmoid()
criterion = nn_MSECriterion()

# Forward-propagation.
a0 = linear.forward(x)
a1 = sigmoid.forward(a0)
loss = criterion.forward(a1, y) # Loss function.

# Backward-propagation.
da1 = criterion.backward(a1, y)
da0 = sigmoid.backward(a0, da1)
dx = linear.backward(x, da0)

gradWeight = linear.gradWeight
gradBias = linear.gradBias

approxGradWeight = np.zeros_like(linear.weight)
approxGradBias = np.zeros_like(linear.bias)

# We will verify here that gradWeights are correct and leave it as an exercise
# to verify the gradBias.
epsilon = 0.0001
for i in range(0, linear.weight.shape[0]):
    for j in range(0, linear.weight.shape[1]):
        # Compute f(w)
        fw = criterion.forward(sigmoid.forward(linear.forward(x)), y) # Loss function.
        # Compute f(w + eps)
        shifted_weight = np.copy(linear.weight)
        shifted_weight[i, j] = shifted_weight[i, j] + epsilon
        shifted_linear = nn_Linear(4, 3)
        shifted_linear.bias = linear.bias
        shifted_linear.weight = shifted_weight
        fw_epsilon = criterion.forward(sigmoid.forward(shifted_linear.forward(x)), y) #
Loss function
        # Compute (f(w + eps) - f(w)) / eps
        approxGradWeight[i, j] = (fw_epsilon - fw) / epsilon

# These two outputs should be similar up to some precision.
print('gradWeight: ' + str(gradWeight))
print('\napproxGradWeight: ' + str(approxGradWeight))
```

```
gradWeight: [[ -3.17948721 -4.3390268  -5.61616626]
 [ -5.16326983 -7.04628285 -9.12027   ]
 [-46.79552973 -63.861574  -82.65844701]
 [ -7.24216532 -9.88333883 -12.7923787 ]]
```

```
approxGradWeight: [[ -3.17947065 -4.33900282 -5.6161322 ]
 [ -5.16322613 -7.04621957 -9.12018013]
 [-46.79189845 -63.85631956 -82.65099267]
 [ -7.2420793  -9.88321427 -12.79220182]]
```

Section 3.4. Playing with the Simple NN [30 pts]

Now, we provide a simple NN model that has an training accuracy around 46%. Your job here is to first run and check the simple NN model, and the experiments it with:

- Try using 3 different numbers of layers for the simple NN, and plot the accuracy as a function of the number of layers.
- Try 4 different numbers of hidden state size, and plot the accuracy as a function of the number you have tried.
- Try different activation function (Sigmoid, ReLU)
- Briefly describe what you've observed in the above experiments

Note that you are free to create your own Simple_NN class for your experiments.

In [45]:

```
class nn_CECriterionWithLogits:
    def forward(self, logits, labels):
        y_indices = np.arange(len(labels))
        self.probs = softmax(logits)
        correct_class_prob = self.probs[y_indices, labels].reshape(-1, 1)
        loss = np.sum(-np.log(correct_class_prob)) / len(correct_class_prob)
        return loss

    def backward(self, labels):
        y_indices = np.arange(len(labels))
        gradOutput = self.probs
        gradOutput[y_indices, labels] -= 1
        return gradOutput / len(labels)

class nn_ReLU:
    def forward(self, x):
        # Forward pass.
        return np.maximum(0, x)

    def backward(self, x, gradOutput):
        # Backward pass
        return np.where(x < 0, 0, np.multiply(1, gradOutput))

def softmax(x):
    exp = np.exp(x - x.max()) # stability trick
    return (exp / np.sum(exp, axis=1))
```

In [46]:

```
class Simple_NN(object):
    def __init__(self, input_size, output_size, hidden_state_size=64):
        self.linear1 = nn_Linear(input_size, hidden_state_size)
        self.activation = nn_ReLU()
        self.linear2 = nn_Linear(hidden_state_size, output_size)

    def forward(self, X):
        self.X = X
        self.x1 = self.linear1.forward(self.X)
        self.a1 = self.activation.forward(self.x1)
        logits = self.linear2.forward(self.a1)
        return logits

    def backward(self, gradOutput):
        dx2 = self.linear2.backward(self.a1, gradOutput)
        da1 = self.activation.backward(self.x1, dx2)
        self.linear1.backward(self.X, da1)

    def update(self, lr):
        self.linear1.weight = self.linear1.weight - lr * (self.linear1.gradWeight)
        self.linear1.bias = self.linear1.bias - lr * (self.linear1.gradBias)
        self.linear2.weight = self.linear2.weight - lr * (self.linear2.gradWeight)
        self.linear2.bias = self.linear2.bias - lr * (self.linear2.gradBias)

    def train(self, X_train, y_train, loss_criterion=nn_CECriterionWithLogits(),
              lr=1e-5, n_iter=1500, batch_size=200):
        loss_hist = []

        for i in range(n_iter):
            batch_indices = np.random.choice(len(X_train), batch_size)
            X_batch = X_train[batch_indices]
            y_batch = y_train[batch_indices]

            logits = self.forward(X_batch)
            loss = loss_criterion.forward(logits, y_batch)
            loss_grad = loss_criterion.backward(y_batch)
            self.backward(loss_grad)
            self.update(lr)

            if (i + 1) % 100 == 0:
                print(f'Iter {i+1}/{n_iter} - loss: {loss} ')
            loss_hist.append(loss)
        return loss_hist

    def predict(self, X):
        return np.asarray(np.argmax(softmax(self.forward(X)), axis=1)).squeeze(1)
```

As the bias is explicitly handled in the MLP code, stripped off the ones we have concatenated to the feature vectors.

In [47]:

```
X_train = X_train[:, :3072]
X_val = X_val[:, :3072]
X_test = X_test[:, :3072]
```

Now, run the simple NN to check the accuracy:

In [48]:

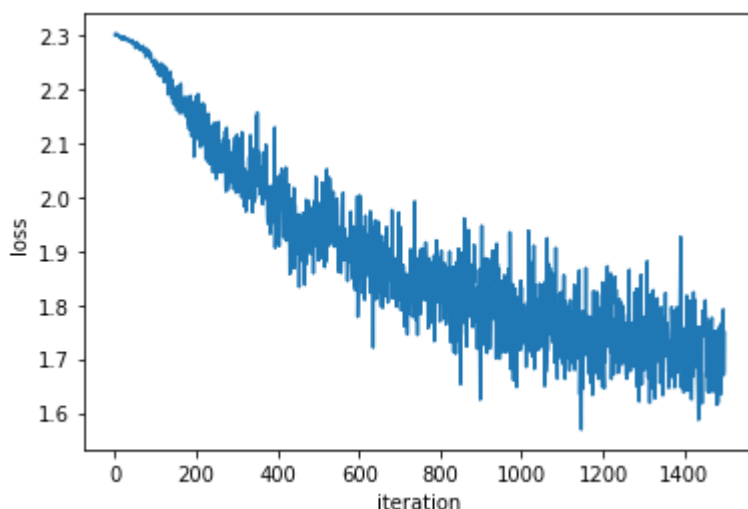
```
model = Simple_NN(X_train.shape[1], np.max(y_train)+1, hidden_state_size=64)
n_iter = 1500
batch_size = 200
loss_hist = model.train(X_train, y_train, lr=1e-4)
```

```
Iter 100/1500 - loss: 2.2549310605513306
Iter 200/1500 - loss: 2.1192539305843128
Iter 300/1500 - loss: 2.08162492366329
Iter 400/1500 - loss: 1.983874124496117
Iter 500/1500 - loss: 1.8944739352046998
Iter 600/1500 - loss: 1.8838867337991
Iter 700/1500 - loss: 1.89134797202472
Iter 800/1500 - loss: 1.7694129610363887
Iter 900/1500 - loss: 1.7462254624812352
Iter 1000/1500 - loss: 1.7769940233856938
Iter 1100/1500 - loss: 1.7971673633706207
Iter 1200/1500 - loss: 1.842050386430194
Iter 1300/1500 - loss: 1.7070660409921794
Iter 1400/1500 - loss: 1.63774565650576
Iter 1500/1500 - loss: 1.7493674416951959
```

In [49]:

```
plt.plot(loss_hist);
plt.ylabel('loss');
plt.xlabel('iteration');
predict = model.predict(X_train)
print('The training accuracy is given by : %f' % (get_acc(predict, y_train)))
predict = model.predict(X_val)
print('The training accuracy is given by : %f' % (get_acc(predict, y_val)))
```

The training accuracy is given by : 39.928571
The training accuracy is given by : 41.200000



Section 3.4.1. Different numbers of layers [12 pts]

Try building 3 simple NN with different numbers of layers. Plot the training/validation/test accuracy as a function of the numbers of layers.

In [56]:

```
# Type your code here
# Linear activation function
class Linear(object):
    def forward(self, x):
        return x

    def backward(self, x, gradOutput):
        return gradOutput

# min max normalization
class MinMaxScaler(object):
    def __init__(self):
        self.max = 0.0
        self.min = 0.0

    def fit(self, data):
        self.max = np.max(data, axis=0)
        self.min = np.min(data, axis=0)

    def transform(self, data):
        return (data - self.min) / (self.max - self.min)

    def fit_transform(self, data):
        self.max = np.max(data, axis=0)
        self.min = np.min(data, axis=0)
        return (data - self.min) / (self.max - self.min)

    def inverse(self, data):
        return np.multiply(data, self.max - self.min) + self.min

class AdaDelta(object):
    def __init__(self, param):
        self.sqr = np.zeros_like(param).A
        self.delta = np.zeros_like(param).A

    def update(self, grad, rho=0.9999, epsilon = 1e-5):
        grad = np.asarray(grad)
        self.sqr = rho * self.sqr + (1.0 - rho) * np.square(grad)
        cur_delta = np.sqrt(self.delta + epsilon)
        cur_delta = cur_delta / np.sqrt(self.sqr + epsilon)
        cur_delta = cur_delta * grad
        self.delta = rho * self.delta + (1.0 - rho) * cur_delta * cur_delta
        return cur_delta

# a simple dense layer class for MLP
class DenseLayer(object):
    def __init__(self, input_size, output_size, act=nn_ReLU, optim=AdaDelta):
        self.input_size = input_size
        self.output_size = output_size
        self.linear = nn_Linear(input_size, output_size)
        self.activation = act()

        # Xavier initialization
        Xavier_scale = np.sqrt(2 / (self.input_size + self.output_size))
        self.linear.weight = np.matlib.randn(input_size, output_size) * Xavier_scale
        self.linear.bias = np.matlib.randn((1, output_size)) * Xavier_scale * 0.1

        # optimizer
        self.optim_w = optim(self.linear.weight)
```

```

        self.optim_b = optim(self.linear.bias)

    def forward(self, X, safe_bound=1e3):
        if np.isnan(np.sum(X)):
            self.X = np.random.rand(X.shape)
        else:
            self.X = X

        self.x = self.linear.forward(self.X)
        self.a = self.activation.forward(self.x)

        while np.isnan(np.sum(self.a)) or np.mean(np.abs(self.a)) > safe_bound:
            self.linear.weight *= 0.5
            self.linear.bias *= 0.5
            self.x = self.linear.forward(self.X)
            self.a = self.activation.forward(self.x)
        return self.a

    def backward(self, gradOutput):
        da = self.activation.backward(self.x, gradOutput)
        return self.linear.backward(self.X, da)

    def update(self, lr=1e-5, reg=0.0, clip=1e4):
        grad_w = np.clip(self.linear.gradWeight, -clip, clip)
        grad_b = np.clip(self.linear.gradBias, -clip, clip)
        self.linear.weight -= self.optim_w.update(self.linear.gradWeight, epsilon=lr)
        self.linear.weight -= reg * self.linear.weight
        self.linear.bias -= self.optim_b.update(self.linear.gradBias, epsilon=lr)

```

MLP model

```

class MLP(object):
    def __init__(self, input_size, output_size, hidden_state_size=64, depth=1, act=nn.ReLU):
        self.input_size = input_size
        self.output_size = output_size
        self.depth = depth
        self.hidden_state_size = hidden_state_size

        # input layer
        self.input_layer = DenseLayer(
            input_size,
            hidden_state_size,
            act
        )

        # hidden layers
        self.hidden_layers = [DenseLayer(
            hidden_state_size,
            hidden_state_size,
            act
        ) for _ in range(depth-1)]

        # output layer
        self.output_layer = DenseLayer(
            hidden_state_size,
            output_size,
            Linear
        )

        # min-max normalization

```

```

        self.min_max_scaler = MinMaxScaler()

    def forward(self, x):
        x = self.input_layer.forward(x)
        for layer in self.hidden_layers:
            x = layer.forward(x)
        logits = self.output_layer.forward(x)
        return logits

    def backward(self, gradOutput):
        dx = self.output_layer.backward(gradOutput)
        for layer in reversed(self.hidden_layers):
            dx = layer.backward(dx)
        self.input_layer.backward(dx)

    def update(self, lr):
        self.input_layer.update(lr)
        for layer in self.hidden_layers:
            layer.update(lr)
        self.output_layer.update(lr)

    def train(
        self,
        X_train,
        y_train,
        loss_criterion=nn_CECriterionWithLogits(),
        lr=1e-5,
        n_iter=1500,
        batch_size=200,
        verbose=True,
        print_iter=100
    ):
        loss_hist = []

        # normalization
        X_train = self.min_max_scaler.fit_transform(X_train)

        for i in range(n_iter):
            batch_indices = np.random.choice(len(X_train), batch_size)
            X_batch = X_train[batch_indices]
            y_batch = y_train[batch_indices]

            logits = self.forward(X_batch)
            loss = loss_criterion.forward(logits, y_batch)
            loss_grad = loss_criterion.backward(y_batch)
            self.backward(loss_grad)
            self.update(lr)

            if (i + 1) % print_iter == 0 and verbose:
                print(f'Iter {i+1} / {n_iter} - loss: {loss} ')
            loss_hist.append(loss)
        return loss_hist

    def predict(self, X):
        X = self.min_max_scaler.transform(X)
        return np.asarray(np.argmax(softmax(self.forward(X)), axis=1)).squeeze(1)

```

```

models = [
    MLP(X_train.shape[1], np.max(y_train)+1, hidden_state_size=64, depth=i, act=nn_Sigmoid) for i in range(1, 4)
]

```

```

losses = []

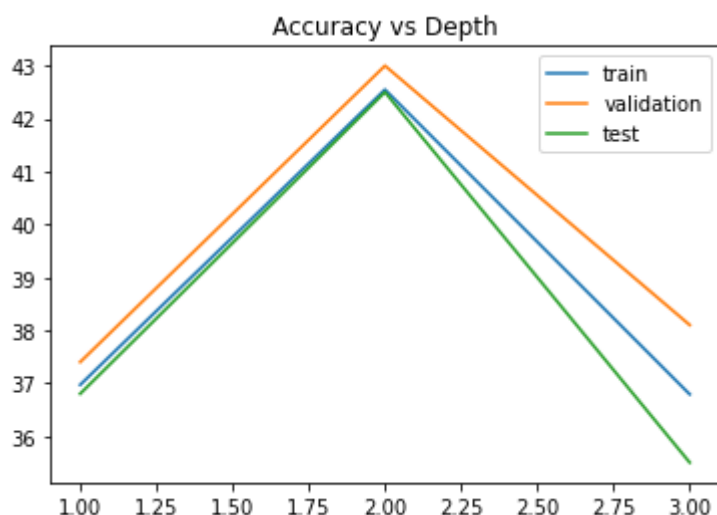
# train the models
for i in range(len(models)):
    model = models[i]
    n_iter = 1500
    batch_size = 200
    losses.append(model.train(X_train, y_train, lr=1e-5, n_iter=n_iter, batch_size=batch_size, verbose=False))

test_accuracy, train_accuracy, val_accuracy = [], [], []
for i in range(len(losses)):
    print('Training model depth', i + 1)
    predict = models[i].predict(X_train)
    print('The training accuracy is: %f' % (get_acc(predict, y_train)))
    train_accuracy.append(get_acc(predict, y_train))
    predict = models[i].predict(X_val)
    print('The validation accuracy is: %f' % (get_acc(predict, y_val)))
    val_accuracy.append(get_acc(predict, y_val))
    predict = models[i].predict(X_test)
    print('The test accuracy is: %f' % (get_acc(predict, y_test)))
    test_accuracy.append(get_acc(predict, y_test))
plt.show()

train_accuracy = np.array(train_accuracy)
val_accuracy = np.array(val_accuracy)
test_accuracy = np.array(test_accuracy)
plt.plot(np.arange(1, 4), train_accuracy)
plt.plot(np.arange(1, 4), val_accuracy)
plt.plot(np.arange(1, 4), test_accuracy)
plt.legend(['train', 'validation', 'test'])
plt.title('Accuracy vs Depth')
plt.show()

```

Training model depth 1
The training accuracy is: 36.969388
The validation accuracy is: 37.400000
The test accuracy is: 36.800000
Training model depth 2
The training accuracy is: 42.548980
The validation accuracy is: 43.000000
The test accuracy is: 42.500000
Training model depth 3
The training accuracy is: 36.789796
The validation accuracy is: 38.100000
The test accuracy is: 35.500000



Section 3.4.2. Different numbers of hidden state size [7 pts]

Try 4 different numbers of hidden state size, and plot the accuracy as a function of the number you have tried. We recommend you to test numbers like 8, 16, 32, 64, 128, 256. Plot the training/validation/test accuracy as a function of the hidden state size your picked.

In [60]:

```
# Type your code here
# [1, 2, 3] layers
hidden_sizes = [2**i for i in range(3, 9)]
print('hidden sizes =', hidden_sizes)

models = [
    MLP(X_train.shape[1], np.max(y_train)+1, hidden_state_size=2**i, depth=1, act=nn_Si
gmoid) for i in range(3, 9)
]

losses = []

# train the models
for i in range(len(models)):
    model = models[i]
    n_iter = 1500
    batch_size = 200
    print('Training model hidden state size', hidden_sizes[i])
    losses.append(model.train(X_train, y_train, lr=1e-5, n_iter=n_iter, batch_size=batch
h_size, verbose=False))
```

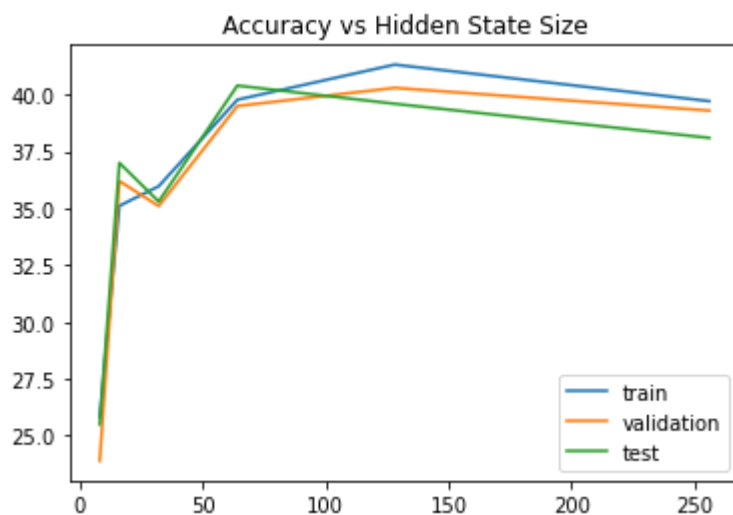
```
hidden sizes = [8, 16, 32, 64, 128, 256]
Training model hidden state size 8
Training model hidden state size 16
Training model hidden state size 32
Training model hidden state size 64
Training model hidden state size 128
Training model hidden state size 256
```

In [67]:

```
test_acc, train_acc, val_acc = [], [], []
for i in range(len(losses)):
    print('Training model hidden state size', hidden_sizes[i])
    predict = models[i].predict(X_train)
    print('The training accuracy is given by : %f' % (get_acc(predict, y_train)))
    train_acc.append(get_acc(predict, y_train))
    predict = models[i].predict(X_val)
    print('The validation accuracy is given by : %f' % (get_acc(predict, y_val)))
    val_acc.append(get_acc(predict, y_val))
    predict = models[i].predict(X_test)
    print('The test accuracy is given by : %f' % (get_acc(predict, y_test)))
    test_acc.append(get_acc(predict, y_test))

train_acc = np.array(train_acc)
val_acc = np.array(val_acc)
test_acc = np.array(test_acc)
plt.plot(np.array(hidden_sizes), train_acc)
plt.plot(np.array(hidden_sizes), val_acc)
plt.plot(np.array(hidden_sizes), test_acc)
plt.legend(['train', 'validation', 'test'])
plt.title('Accuracy vs Hidden State Size')
plt.show()
```

Training model hidden state size 8
The training accuracy is given by : 25.820408
The validation accuracy is given by : 23.900000
The test accuracy is given by : 25.500000
Training model hidden state size 16
The training accuracy is given by : 35.108163
The validation accuracy is given by : 36.200000
The test accuracy is given by : 37.000000
Training model hidden state size 32
The training accuracy is given by : 35.971429
The validation accuracy is given by : 35.100000
The test accuracy is given by : 35.300000
Training model hidden state size 64
The training accuracy is given by : 39.777551
The validation accuracy is given by : 39.500000
The test accuracy is given by : 40.400000
Training model hidden state size 128
The training accuracy is given by : 41.322449
The validation accuracy is given by : 40.300000
The test accuracy is given by : 39.600000
Training model hidden state size 256
The training accuracy is given by : 39.714286
The validation accuracy is given by : 39.300000
The test accuracy is given by : 38.100000



Section 3.4.3. Different activation functions [7 pts]

Try both ReLU and Sigmoid activation, and report the training/validation/test accuracy.

In [68]:

```
# Type your code here
model = MLP(X_train.shape[1], np.max(y_train)+1, hidden_state_size=64, depth=1, act=nn_
ReLU)
n_iter = 1500
batch_size = 200
loss_hist = model.train(X_train, y_train, lr=1e-5, n_iter=n_iter, batch_size=batch_size
)

print('Activation: ReLU')
predict = model.predict(X_train)
print('The training accuracy is: %f' % (get_acc(predict, y_train)))
predict = model.predict(X_val)
print('The validation accuracy is: %f' % (get_acc(predict, y_val)))
```

```
Iter 100 / 1500 - loss: 2.307157012953204
Iter 200 / 1500 - loss: 2.3069803210148216
Iter 300 / 1500 - loss: 2.2998088881374894
Iter 400 / 1500 - loss: 2.3081824123897716
Iter 500 / 1500 - loss: 2.300799169709764
Iter 600 / 1500 - loss: 2.3041243022358997
Iter 700 / 1500 - loss: 2.302512112522636
Iter 800 / 1500 - loss: 2.304031772748081
Iter 900 / 1500 - loss: 2.3017270347147223
Iter 1000 / 1500 - loss: 2.3048526513655827
Iter 1100 / 1500 - loss: 2.305082308587993
Iter 1200 / 1500 - loss: 2.300277624715537
Iter 1300 / 1500 - loss: 2.298333319367239
Iter 1400 / 1500 - loss: 2.3083317876065523
Iter 1500 / 1500 - loss: 2.305099704872987
Activation: ReLU
The training accuracy is: 9.995918
The validation accuracy is: 10.200000
```

In [69]:

```
model = MLP(X_train.shape[1], np.max(y_train)+1, hidden_state_size=64, depth=1, act=nn_
Sigmoid)
n_iter = 1500
batch_size = 200
loss_hist = model.train(X_train, y_train, lr=1e-5, n_iter=n_iter, batch_size=batch_size
)

print('Activation: Sigmoid')
predict = model.predict(X_train)
print('The training accuracy is: %f' % (get_acc(predict, y_train)))
predict = model.predict(X_val)
print('The validation accuracy is: %f' % (get_acc(predict, y_val)))
```

```
Iter 100 / 1500 - loss: 2.1936435665207212
Iter 200 / 1500 - loss: 2.134921520748347
Iter 300 / 1500 - loss: 2.1085138231802354
Iter 400 / 1500 - loss: 2.05444420494505
Iter 500 / 1500 - loss: 1.8913715983412587
Iter 600 / 1500 - loss: 1.8791890819346013
Iter 700 / 1500 - loss: 1.781075076656003
Iter 800 / 1500 - loss: 1.685927581556403
Iter 900 / 1500 - loss: 1.6067841328591295
Iter 1000 / 1500 - loss: 1.733706107717817
Iter 1100 / 1500 - loss: 1.6879174571305884
Iter 1200 / 1500 - loss: 1.811619655287684
Iter 1300 / 1500 - loss: 1.729500998671989
Iter 1400 / 1500 - loss: 1.7281986493451
Iter 1500 / 1500 - loss: 1.7474059518985994
Activation: Sigmoid
The training accuracy is: 40.565306
The validation accuracy is: 42.700000
```

Section 3.4.4. What did you see? [4 pts]

Describe what you see in the experiments above in 5-6 sentences.

Your answer:

1. With the deepening of the neural network, the risk of encountering gradient disappearance and explosion gradient will be higher.
2. The test accuracy will increase as the hidden state increases, but if the hidden state is too much, it might cause overfitting and the accuracy might decrease.
3. ReLU has a higher chance of explosion gradient and it behaves close to a linear unit.
4. Sigmoid activation convergence prevents exploding gradient.