Yixuan Qiu, Nelson Wu

Fall 2020

CS 343: Neural Networks

Project 2: Multi-layer Perceptrons

In [1]:
```
import os
import random
import numpy as np
import matplotlib.pyplot as plt

# for obtaining the STL-dataset
import load_stl10_dataset

# for preprocessing dataset
import preprocess_data

# Set the color style so that Professor Layton can see your plots
plt.style.use(['seaborn-colorblind', 'seaborn-darkgrid'])
# Make the font size larger
plt.rcParams.update({'font.size': 20})

# Turn off scientific notation when printing
np.set_printoptions(suppress=True, precision=3)

# Automatically reload external modules
%load_ext autoreload
%autoreload 2
```

Overview

In this project, you will implement Multilayer Perception (MLP) networks to classify image data belonging to one of 10 classes from the STL-10 dataset.

You will create two networks that use the softmax activation function and cross-entropy loss:

- One without a hidden layer (**Task 2**: `softmax_layer.ipynb`) and
- One with a hidden layer (**Task 3**: `mlp.ipynb`)

That is, before you implement the MLP with hidden layer, you will create a simpler single layer network (similar architecture to ADALINE) to test your loss and activation function code.

Goals

You will get experience working with:

- a large "real" image dataset
- batch vs stochastic training
- regularization
- multi-class data classification
- network weight visualization

Task 1: Download and preprocess datasets

STL-10

More info: https://cs.stanford.edu/~acoates/stl10/

Properties:

- 10 images classes: airplane, bird, car, cat, deer, dog, horse, monkey, ship, truck
- 5000 images.
- 96x96 resolution (we will work with them converted to 32x32 for rapid development and debugging purposes)

1a. Download the STL-10 dataset, convert to 32x32 images

This data acquistion script should "just work". You're welcome to take a peek at the script to see where the data is going, but you shouldn't need to tinker with it. It needs to download the dataset from the internet, which might take a few minutes depending on your internet speed. I set up the script so that it caches the dataset locally in your project folder, so you should only need to download it once. There is a progress percentage indicator.

You should see the following output:

```
Downloading stl10_binary.tar.gz 100.00%Downloaded stl10_binary.tar.gz
Images are: (5000, 96, 96, 3)
Labels are: (5000,)
Resizing 5000 images to 32x32...Done!
Saving Numpy arrays the images and labels to ./numpy...Done!
```

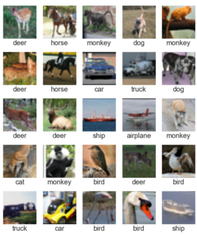
In [2]:
```
# Download the dataset from the internet, convert it to Numpy ndarray, resize to 32x32
# cache it locally on your computer for faster loading next time.
stl_imgs, stl_labels = load_stl10_dataset.load()
```

```
Downloading stl10_binary.tar.gz 100.00%Downloaded stl10_binary.tar.gz
Images are: (5000, 96, 96, 3)
Labels are: (5000,)
Resizing 5000 images to 32x32...Done!
Saving Numpy arrays the images and labels to ./numpy...Done!
```

1b. Take a look at a sample of the images and their associated classes in the dataset

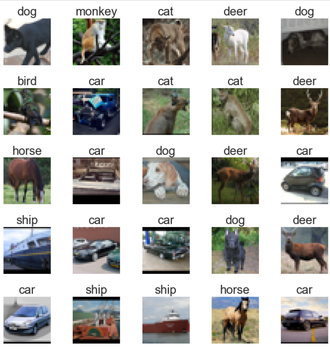
These image-class pairs are what you will train your network on.

In the cell below, create a 5x5 plot showing 25 **randomly selected** images from the dataset. Label each image with its associated class. An example output is included below (*your image will differ due to randomness*). **Do a sanity check!** Does the class look correct for each image?



In [31]:
```
# Load in the string names for each class
classes = np.loadtxt(os.path.join('data', 'stl10_binary', 'class_names.txt'), dtype=str)
```

In [32]:
```
# Your 5x5 image grid plotting code here
fig, axes = plt.subplots(5, 5, figsize = (10, 10))

# Iterate each subplot and plot the graph
for i in range(5):
    for j in range(5):
        index = random.randint(0, 5000)
        axes[i, j].imshow(stl_imgs[index])
        axes[i, j].set_title(classes[stl_labels[index] - 1])
        axes[i, j].axis('off')

fig.tight_layout()
plt.show()
```



1c. Preprocess the data

Preprocessing data is an important part of working with neural networks.

Fill in the function `preprocess_stl` in `preprocess_data.py`, implementing the 4 listed goals. Run the below test code to get the desired print out.

In [33]:
```
# Preprocess image pixel values for the MLP net
stl_imgs_pp, stl_labels_pp = preprocess_data.preprocess_stl(stl_imgs, stl_labels)

print(f'stl_imgs dtype is {stl_imgs_pp.dtype} and it should be float64')
print(f'stl_imgs max is {np.max(stl_imgs_pp[:, 1:]):.3f} and it should be 0.668')
print(f'stl_imgs shape is {stl_imgs_pp.shape} and it should be (5000, 3072)')
print(f'stl_labels span {stl_labels_pp.min()}->{stl_labels_pp.max()} and it should be 0->9')
```

```
stl_imgs dtype is float64 and it should be float64
stl_imgs max is 0.934 and it should be 0.668
stl_imgs shape is (5000, 3072) and it should be (5000, 3072)
stl_labels span 0->9 and it should be 0->9
```

1d. Create train, validation, test, and dev splits

Complete the `create_splits` function in `preprocess_data.py` to divvy up the dataset into train/test/validation/development subsets, following the two main goals lists. You should obtain a printout as follows:

```
Train data shape:  (3500, 3072)
Train labels shape:  (3500,)
Test data shape:  (500, 3072)
Test labels shape:  (500,)
Validation data shape:  (500, 3072)
Validation labels shape:  (500,)
dev data shape:  (500, 3072)
dev labels shape:  (500,)
```

In [34]:
```
x_train, y_train, x_test, y_test, x_val, y_val, x_dev, y_dev = preprocess_data.create_splits(stl_imgs_pp, stl_labels_p
p)

print ('Train data shape: ', x_train.shape)
print ('Train labels shape: ', y_train.shape)
print ('Test data shape: ', x_test.shape)
print ('Test labels shape: ', y_test.shape)
print ('Validation data shape: ', x_val.shape)
print ('Validation labels shape: ', y_val.shape)
print ('dev data shape: ', x_dev.shape)
print ('dev labels shape: ', y_dev.shape)
```

```
Train data shape:  (3500, 3072)
Train labels shape:  (3500,)
Test data shape:  (500, 3072)
Test labels shape:  (500,)
Validation data shape:  (500, 3072)
Validation labels shape:  (500,)
dev data shape:  (500, 3072)
dev labels shape:  (500,)
```

1e. Automate loading and preprocessing STL-10

Complete the `load_stl10` function in `preprocess_data.py` to do all the above preprocessing steps. You can (selectively) copy and paste! You should obtain a printout as follows:

```
Found cached numpy arrays the images and labels. Loading them...
Images are: (5000, 32, 32, 3)
Labels are: (5000,)
Train data shape:  (3000, 3072)
Train labels shape:  (3000,)
Test data shape:  (750, 3072)
Test labels shape:  (750,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
dev data shape:  (250, 3072)
dev labels shape:  (250,)
```

Note: The below test code uses a different split to help exercise your code.

In [35]:
```
x_train, y_train, x_test, y_test, x_val, y_val, x_dev, y_dev = preprocess_data.load_stl10(n_train_samps=3000, n_test_s
amps=750, n_valid_samps=1000, n_dev_samps=250)

print ('Train data shape: ', x_train.shape)
print ('Train labels shape: ', y_train.shape)
print ('Test data shape: ', x_test.shape)
print ('Test labels shape: ', y_test.shape)
print ('Validation data shape: ', x_val.shape)
print ('Validation labels shape: ', y_val.shape)
print ('dev data shape: ', x_dev.shape)
print ('dev labels shape: ', y_dev.shape)
```

```
Found cached numpy arrays the images and labels. Loading them...
Images are: (5000, 32, 32, 3)
Labels are: (5000,)
Train data shape:  (3000, 3072)
Train labels shape:  (3000,)
Test data shape:  (750, 3072)
Test labels shape:  (750,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
dev data shape:  (250, 3072)
dev labels shape:  (250,)
```

Nathan Mu, Vincent Zhu
Fall 2020
CS 230: Deep Neural Networks
Project 2: Multi-Input Paraphrasing

```
import os
import numpy as np
import keras as kr
import keras.backend as K
import tensorflow as tf
from keras.models import Sequential, Model, load_model
from keras.layers import *
import keras.preprocessing.text as kpt
import keras.preprocessing.sequence as kps

# set the random seed so that results are reproducible
np.random.seed(1)

# fixed GPU memory allocation for compatibility
config = tf.ConfigProto(gpu_options=tf.GPUOptions(allow_growth=True))
sess = tf.Session(config=config)
K.set_session(sess)

# disable verbose output for cleaner logs
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
tf.logging.set_verbosity(tf.logging.ERROR)
print("Using Keras version", kr.__version__)
print("Using TensorFlow version", tf.__version__)
```

```
Using Keras version 2.2.4
Using TensorFlow version 1.12.0
```

Task 2: Implement single-layer network to test softmax activation and cross-entropy loss

Implement the forward and loss for a single-layer network with softmax activation and cross-entropy loss in order to test numpy vs. keras implementations.

2a. Load in preprocessed STL-10 data

Copy over preprocessing functions from notebook for STL-10 data in the following cells:

- OHE training labels
- OHE test labels
- Split validation set data
- Grid sample for development

Note that the data files you need to be able to operate under `load_logs` and the LAST 10 lessons in the dir or file `load_lessons`. These are the variable names assigned to the lessons.

```
in [1]: X_train, y_train, X_test, y_test = load_lessons('stl_data', 'stl_data')
        print("X_train.shape", X_train.shape, "y_train.shape", y_train.shape, "X_test.shape", X_test.shape, "y_test.shape", y_test.shape)
        # OHE labels
        num_classes = 10
        y_train = kr.utils.to_categorical(y_train, num_classes)
        y_test = kr.utils.to_categorical(y_test, num_classes)
        print("After OHE y_train.shape", y_train.shape, "y_test.shape", y_test.shape)
        # Split validation set
        X_val = X_train[-1000:]
        y_val = y_train[-1000:]
        X_train = X_train[:-1000]
        y_train = y_train[:-1000]
        print("After validation split X_train.shape", X_train.shape, "y_train.shape", y_train.shape, "X_val.shape", X_val.shape, "y_val.shape", y_val.shape)
        # Grid sample for development
        X_train = X_train[::10]
        y_train = y_train[::10]
        print("After grid sample X_train.shape", X_train.shape, "y_train.shape", y_train.shape)
```

```
X_train.shape (5000, 96, 96, 3) y_train.shape (5000,) X_test.shape (8000, 96, 96, 3) y_test.shape (8000,)
After OHE y_train.shape (5000, 10) y_test.shape (8000, 10)
After validation split X_train.shape (4000, 96, 96, 3) y_train.shape (4000, 10) X_val.shape (1000, 96, 96, 3) y_val.shape (1000, 10)
After grid sample X_train.shape (400, 96, 96, 3) y_train.shape (400, 10)
```

2b. Implement softmax activation function

```
in [2]: # implement softmax activation
        def softmax(x):
            '''
            Softmax activation function
            x: input to softmax of shape (batch_size, dim)
            Returns: output of softmax transformation applied to last dimension of x
            '''
            e = K.exp(x - K.max(x, axis=-1, keepdims=True))
            s = K.sum(e, axis=-1, keepdims=True)
            return e / s
```

2c. Implement the following functions

In `utilities_layer.py`, implement the following methods in the class name `DenseNaive`:

- `__init__()`
- `build()`
- `call()`
- `compute_output_shape()`
- `get_config()`
- `activation(x, kwargs)`: `activation(x, kwargs)` calls a, let the "list" of values and them and f. output neurons (one per input class). `f(x,)` is the activation values of each output neuron. Since this is softmax, it is the probability that a given input belongs to the class. `kwargs` is the output neuron.
- `loss(self, y_true, y_pred)`: `loss(self, y_true, y_pred)` is the loss function. `y_true` is the ground truth and `y_pred` is the output of `activation(x, kwargs)`.

`loss()`, `f()`, `activation()` should return the mean of the losses in the batch. The batch size is `K.int_shape(y_pred)[0]`. Since `loss()` is called before `activation()`, `f()` is called after `activation()`. The batch size is `K.int_shape(y_pred)[0]`. Since `loss()` is called before `activation()`, `f()` is called after `activation()`. The batch size is `K.int_shape(y_pred)[0]`.

Hint: `f()` is the `f` in `activation(x, kwargs)`.

Hint: `activation(x, kwargs)` is the `activation` in `Dense(units, activation=activation)`.

The following cells will test your implementation. Verify that `f()` returns an array with length 10, `activation()` returns a tensor with shape `(batch_size, 10)`, and `loss()` returns a scalar.

```
in [3]: # implement DenseNaive
        class DenseNaive(Layer):
            '''
            Dense layer implementation
            '''
            def __init__(self, units, activation=None, kwargs={}, input_shape=None):
                '''
                units: output dimension of layer
                activation: activation function to apply to output of layer
                kwargs: dictionary of keyword arguments for activation function
                input_shape: shape of input to layer
                '''
                super(DenseNaive, self).__init__(kwargs)
                self.units = units
                self.activation = activation
                self.kwargs = kwargs
                self.input_shape = input_shape
            def __repr__(self):
                return "DenseNaive(units=%d, activation=%s, kwargs=%s, input_shape=%s)" % (self.units, self.activation, self.kwargs, self.input_shape)
            def __str__(self):
                return "DenseNaive(units=%d, activation=%s, kwargs=%s, input_shape=%s)" % (self.units, self.activation, self.kwargs, self.input_shape)
            def build(self, input_shape):
                '''
                Keras layer initialization.
                input_shape: shape of input to this layer
                '''
                self.input_shape = input_shape
                self.kernel = self.add_weight(name='kernel', shape=(input_shape[1], self.units),
                                              initializer='uniform',
                                              trainable=True)
                self.bias = self.add_weight(name='bias', shape=(self.units,),
                                            initializer='uniform',
                                            trainable=True)
                super(DenseNaive, self).build(input_shape)
            def call(self, x):
                '''
                Keras layer call.
                x: input to layer
                '''
                return self.activation(x, self.kernel, self.bias, self.kwargs)
            def compute_output_shape(self, input_shape):
                '''
                Keras layer output shape.
                input_shape: shape of input to this layer
                '''
                return (input_shape[0], self.units)
            def get_config(self):
                '''
                Keras layer configuration.
                '''
                base = super(DenseNaive, self).get_config()
                return dict(list(base.items()) + [('units', self.units), ('activation', self.activation), ('kernel_shape', self.kernel.shape), ('bias_shape', self.bias.shape), ('kwargs', self.kwargs), ('input_shape', self.input_shape)])
            def loss(self, y_true, y_pred):
                '''
                Keras loss function.
                y_true: tensor of ground truth values
                y_pred: tensor of predicted values
                '''
                return K.mean(K.sum(K.square(y_true - y_pred), axis=-1))
            def f(self, x, kernel, bias, kwargs):
                '''
                Dense layer function.
                x: input to layer
                kernel: weight matrix
                bias: bias vector
                kwargs: dictionary of keyword arguments for activation function
                '''
                return K.dot(x, kernel) + bias
```

Test the `loss()`, `loss_np`, softmax activation functions

```
in [4]: inputDense = Input(shape=(10,))
        dense = DenseNaive(units=10, activation='softmax')(inputDense)
        model = Model(inputs=inputDense, outputs=dense)
        model.compile('adam', dense.loss)
        print("Keras softmax", model.predict(np.ones((1,10,))))
        print("Numpy softmax", softmax(np.ones((1,10,))).eval())
```

```
Keras softmax [[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]]
Numpy softmax [[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]]
```

Test the `activation()` function

```
in [5]: model = load_model('stl.h5', {'DenseNaive':DenseNaive})
        y_pred = model.predict(X_train[:1])
        print("Keras loss", np.sum(np.square(y_train[:1] - y_pred), axis=-1))
        print("Numpy loss", np.sum(np.square(y_train[:1] - y_pred), axis=-1).eval())
```

```
Keras loss [0.9000001]
Numpy loss [0.9000001]
```

Your code for softmax should look like:

```
[[0. 0. 0. 0. 0.
  0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.
  0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.
  0. 0. 0. 0. 0.]]
```

Your code for softmax should look like:

```
[[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
 [0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
 [0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]]
```

Test the `loss`, `loss_np`, softmax activation functions

```
in [6]: inputDense = Input(shape=(10,))
        dense = DenseNaive(units=10, activation='softmax')(inputDense)
        model = Model(inputs=inputDense, outputs=dense)
        model.compile('adam', dense.loss)
        print("Keras softmax", model.predict(np.ones((1,10,))))
        print("Numpy softmax", softmax(np.ones((1,10,))).eval())
```

```
Keras softmax [[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]]
Numpy softmax [[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]]
```

Test the `loss()` function

```
in [7]: y_pred = model.predict(X_train[:1])
        print("Keras loss", np.sum(np.square(y_train[:1] - y_pred), axis=-1))
        print("Numpy loss", np.sum(np.square(y_train[:1] - y_pred), axis=-1).eval())
```

```
Keras loss [0.9000001]
Numpy loss [0.9000001]
```

Your loss should look like:

```
Keras loss [0.9000001]
Numpy loss [0.9000001]
```

Your loss should look like:

```
Keras loss [0.9000001]
Numpy loss [0.9000001]
```

2d. Train your function

```
in [8]: # compile model
        model.compile('adam', dense.loss)
        # fit model
        model.fit(X_train, y_train, batch_size=32, epochs=100, verbose=1, validation_data=(X_val, y_val))
        # save model
        model.save('stl.h5')
        # load model
        model = load_model('stl.h5', {'DenseNaive':DenseNaive})
        # predict
        y_pred = model.predict(X_train)
        print("Keras loss", np.sum(np.square(y_train - y_pred), axis=-1))
        print("Numpy loss", np.sum(np.square(y_train - y_pred), axis=-1).eval())
```

```
Keras loss [0.9000001 0.9000001 0.9000001 0.9000001 0.9000001 0.9000001
 
```

```
coarse search best lr:  0.07999999999999999

coarse search best reg:  0.0

coarse search best bs:  32
```

In [63]:
```
losses = []
lrs = []
regs = []
bss = []

def fine_search(lr_low, lr_high, lr_step, reg_low, reg_high, reg_step, bs_low, bs_high, bs_step):

    for bs in np.arange(bs_low, bs_high + bs_step, bs_step):
        for lr in np.arange(lr_low, lr_high + lr_step, lr_step):
            for reg in np.arange(reg_low, reg_high + reg_step, reg_step):
                softmaxNet = SoftmaxLayer(10)
                loss = softmaxNet.fit(x_train, y_train, n_epochs=10, lr=lr, mini_batch_sz=bs, reg=reg, verbose=0)
                losses.append(loss[-1])
                lrs.append(lr)
                regs.append(reg)
                bss.append(bs)
                print(f"lr: {lr:.4f}  reg: {reg}  bs: {bs}    loss: {loss[-1]}")
```

In [64]:
```
import time
start = time.time()

fine_search(lr_low=0.07, lr_high=0.08, lr_step=0.001, reg_low=0, reg_high=0, reg_step=1, bs_low=16, bs_high=32, bs_ste
p=2)

end = time.time()
print("\ntime elapsed:", end - start)
```

```
lr: 0.0700  reg: 0  bs: 16    loss: 1.6890539101629978
lr: 0.0710  reg: 0  bs: 16    loss: 1.7203256602462593
lr: 0.0720  reg: 0  bs: 16    loss: 1.2450775095733388
lr: 0.0730  reg: 0  bs: 16    loss: 0.9375251712001855
lr: 0.0740  reg: 0  bs: 16    loss: 1.1222214295500441
lr: 0.0750  reg: 0  bs: 16    loss: 0.7753604454393708
lr: 0.0760  reg: 0  bs: 16    loss: 1.3823624659521145
lr: 0.0770  reg: 0  bs: 16    loss: 1.3836565201860986
lr: 0.0780  reg: 0  bs: 16    loss: 1.1463627771649498
lr: 0.0790  reg: 0  bs: 16    loss: 1.3469749465762357
lr: 0.0800  reg: 0  bs: 16    loss: 1.0994613192966356
lr: 0.0700  reg: 0  bs: 18    loss: 1.502981754511074
lr: 0.0710  reg: 0  bs: 18    loss: 1.6723379085131136
lr: 0.0720  reg: 0  bs: 18    loss: 0.9055981032870254
lr: 0.0730  reg: 0  bs: 18    loss: 1.228093811339745
lr: 0.0740  reg: 0  bs: 18    loss: 1.1400517619160797
lr: 0.0750  reg: 0  bs: 18    loss: 1.6569045627223253
lr: 0.0760  reg: 0  bs: 18    loss: 1.3499072020226375
lr: 0.0770  reg: 0  bs: 18    loss: 1.4142286284957308
lr: 0.0780  reg: 0  bs: 18    loss: 1.0673130649761577
lr: 0.0790  reg: 0  bs: 18    loss: 1.6768051262515582
lr: 0.0800  reg: 0  bs: 18    loss: 1.0677693409720495
lr: 0.0700  reg: 0  bs: 20    loss: 1.1254321992870466
lr: 0.0710  reg: 0  bs: 20    loss: 1.3897716682277856
lr: 0.0720  reg: 0  bs: 20    loss: 1.262758555562608
lr: 0.0730  reg: 0  bs: 20    loss: 1.0060013070237148
lr: 0.0740  reg: 0  bs: 20    loss: 1.3544644892171l
lr: 0.0750  reg: 0  bs: 20    loss: 1.1816885070765095
lr: 0.0760  reg: 0  bs: 20    loss: 1.504685376814922
lr: 0.0770  reg: 0  bs: 20    loss: 1.2139838109882734
lr: 0.0780  reg: 0  bs: 20    loss: 1.4734164533384886
lr: 0.0790  reg: 0  bs: 20    loss: 1.3533224692207104
lr: 0.0800  reg: 0  bs: 20    loss: 1.2686187659367225
lr: 0.0700  reg: 0  bs: 22    loss: 1.3049183648316305
lr: 0.0710  reg: 0  bs: 22    loss: 1.1829952111977873
lr: 0.0720  reg: 0  bs: 22    loss: 1.2400864241603031
lr: 0.0730  reg: 0  bs: 22    loss: 1.3843456778156658
lr: 0.0740  reg: 0  bs: 22    loss: 1.289003861925893
lr: 0.0750  reg: 0  bs: 22    loss: 1.6702661398662952
lr: 0.0760  reg: 0  bs: 22    loss: 1.200650711868924
lr: 0.0770  reg: 0  bs: 22    loss: 1.2826996912170372
lr: 0.0780  reg: 0  bs: 22    loss: 1.2514894966703174
lr: 0.0790  reg: 0  bs: 22    loss: 1.3650029424605545
lr: 0.0800  reg: 0  bs: 22    loss: 1.2721846033457005
lr: 0.0700  reg: 0  bs: 24    loss: 1.0849084247930119
lr: 0.0710  reg: 0  bs: 24    loss: 1.3905551197856518
lr: 0.0720  reg: 0  bs: 24    loss: 1.5365822855080324
lr: 0.0730  reg: 0  bs: 24    loss: 1.0950710442747875
lr: 0.0740  reg: 0  bs: 24    loss: 1.4395170570661182
lr: 0.0750  reg: 0  bs: 24    loss: 1.3523076937102558
lr: 0.0760  reg: 0  bs: 24    loss: 1.3032450904639794
lr: 0.0770  reg: 0  bs: 24    loss: 1.2201598584711119
lr: 0.0780  reg: 0  bs: 24    loss: 1.3657544819536227
lr: 0.0790  reg: 0  bs: 24    loss: 1.5231664459294033
lr: 0.0800  reg: 0  bs: 24    loss: 1.3505190618274469
lr: 0.0700  reg: 0  bs: 26    loss: 1.4085494184055678
lr: 0.0710  reg: 0  bs: 26    loss: 1.3673983433200527
lr: 0.0720  reg: 0  bs: 26    loss: 1.0496951641686203
lr: 0.0730  reg: 0  bs: 26    loss: 1.1451774648192024
lr: 0.0740  reg: 0  bs: 26    loss: 1.5460016659411655
lr: 0.0750  reg: 0  bs: 26    loss: 1.0622876840100994
lr: 0.0760  reg: 0  bs: 26    loss: 1.4869038959729066
lr: 0.0770  reg: 0  bs: 26    loss: 1.1849551393304254
lr: 0.0780  reg: 0  bs: 26    loss: 1.4722955732707896
lr: 0.0790  reg: 0  bs: 26    loss: 1.1262421032121137
lr: 0.0800  reg: 0  bs: 26    loss: 1.0216675724201 5
lr: 0.0700  reg: 0  bs: 28    loss: 1.3714855549198105
lr: 0.0710  reg: 0  bs: 28    loss: 1.6019053487908632
lr: 0.0720  reg: 0  bs: 28    loss: 1.4102058465330969
lr: 0.0730  reg: 0  bs: 28    loss: 1.1491879411982617
lr: 0.0740  reg: 0  bs: 28    loss: 1.401077348649286
lr: 0.0750  reg: 0  bs: 28    loss: 1.3947056316494957
lr: 0.0760  reg: 0  bs: 28    loss: 1.2246962345071817
lr: 0.0770  reg: 0  bs: 28    loss: 1.3155804011692356
lr: 0.0780  reg: 0  bs: 28    loss: 1.1068023393194354 7
lr: 0.0790  reg: 0  bs: 28    loss: 1.3325045374523423
lr: 0.0800  reg: 0  bs: 28    loss: 1.5512882515839697
lr: 0.0700  reg: 0  bs: 30    loss: 1.5667729062504325
lr: 0.0710  reg: 0  bs: 30    loss: 1.4917094575380565
lr: 0.0720  reg: 0  bs: 30    loss: 1.7300076016470374
lr: 0.0730  reg: 0  bs: 30    loss: 1.4448112287662185
lr: 0.0740  reg: 0  bs: 30    loss: 1.6388289254981685
lr: 0.0750  reg: 0  bs: 30    loss: 1.2484157464964483
lr: 0.0760  reg: 0  bs: 30    loss: 1.2609305771805597
lr: 0.0770  reg: 0  bs: 30    loss: 1.6559207615640839
lr: 0.0780  reg: 0  bs: 30    loss: 1.3389715700115228
lr: 0.0790  reg: 0  bs: 30    loss: 1.2164580339845013
lr: 0.0800  reg: 0  bs: 30    loss: 1.4988764625484001
lr: 0.0700  reg: 0  bs: 32    loss: 1.5745020524459301
lr: 0.0710  reg: 0  bs: 32    loss: 1.2948500356999515
lr: 0.0720  reg: 0  bs: 32    loss: 1.4705583292333557
lr: 0.0730  reg: 0  bs: 32    loss: 1.3262453361755542
lr: 0.0740  reg: 0  bs: 32    loss: 1.6543018448411995
lr: 0.0750  reg: 0  bs: 32    loss: 1.2972569444066393
lr: 0.0760  reg: 0  bs: 32    loss: 1.4292610973980109
lr: 0.0770  reg: 0  bs: 32    loss: 1.4105240356827724
lr: 0.0780  reg: 0  bs: 32    loss: 1.4432216463788805
lr: 0.0790  reg: 0  bs: 32    loss: 1.3075474014381867
lr: 0.0800  reg: 0  bs: 32    loss: 1.1004487210858704

time elapsed: 9760.36714887619
```

In [65]:
```
losses = np.asarray(losses)
min_idx = np.argmin(losses)
print("\n\nlowest loss: ", np.amin(losses))
print("\nfine search best lr: ", lrs[min_idx])
print("\nfine search best reg: ", regs[min_idx])
print("\nfine search best bs: ", bss[min_idx])
```

```
lowest loss:  0.7753604454393708

fine search best lr:  0.07500000000000001

fine search best reg:  0

fine search best bs:  16
```

In [74]:
```
# Compute the accuracy on the training set
softmaxNet = SoftmaxLayer(10)
softmaxNet.fit(x_train, y_train, n_epochs=100, lr=0.075, mini_batch_sz=16, reg=1, verbose=0)
print(softmaxNet.acc_history[-1])
```

```
0.38742857142857146
```

In [75]:
```
# Compute the accuracy on the validation set
softmaxNet = SoftmaxLayer(10)
softmaxNet.fit(x_val, y_val, n_epochs=100, lr=0.075, mini_batch_sz=16, reg=1, verbose=0)
print(softmaxNet.acc_history[-1])
```

```
0.754
```

2f. Evaluate best model on test set

Question 4: Now that you have "good" parameter values recorded, train a new model with the best learning rate, regularization strength, and batch size values. What accuracy do you get on the test set?

In [76]:
```
# Compute the accuracy on the test set
bestNet = SoftmaxLayer(10)
bestNet.fit(x_test, y_test, n_epochs=100, lr=0.075, mini_batch_sz=16, reg=1, verbose=0)
print(bestNet.acc_history[-1])
```

```
0.75
```

Answer 4:

Accuracy: 0.75

2g. Visualize learned weights

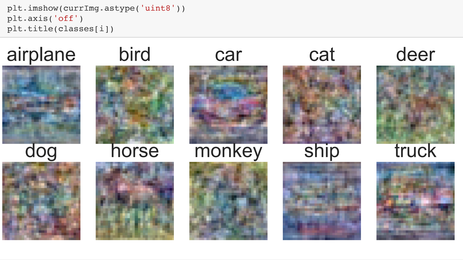
Run the following code that plots the network weights going to each output neuron. If all goes well, you should see something really cool! Include the plot in your submitted project to show me what you got!

Note: the quality of your visualizations will depend on:

- The quality of the hyperparameters that you got via grid search.
- How many epochs that you trained the network before plotting the weights

One extension idea: is to find the combination of the above that result in the best visualizations.

In [77]:
```
# Load in the class names
classes = np.loadtxt(os.path.join('data', 'stl10_binary', 'class_names.txt'), dtype=str)

# We don't care about the bias wt
wts = bestNet.wts
# Reshape the wt vectors into spatial 'image' configurations to visualization
wts = wts.reshape(32, 32, 3, 10)

# Make a large new empty figure/plot
plt.figure(figsize=(10, 4))

# Loop through each output neuron
for i in range(10):
  # Make a 2x5 grid of images
  plt.subplot(2, 5, i+1)

  # Rescale the weights to be between 0 and 255
  currImg = 255.0 * (wts[:, :, :, i].squeeze() - np.min(wts)) / (np.max(wts) - np.min(wts))

  plt.imshow(currImg.astype('uint8'))
  plt.axis('off')
  plt.title(classes[i])
```



In [ ]:

Week 06, Nelson Wu

Fall 2020

CS 230 / Stanford University

Project 1: Multi-layer Perceptrons

Assignment description

- Implement a multi-layer perceptron classifier from scratch
- Submit code, zip file with all of Google Classroom. Includes:
  - `model.ipynb` notebook
  - `utils.py` utilities
  - Data files under `./data/`
- Write a short report (see questions)

```
In [1]: %load_ext autoreload
        %autoreload 2
        %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        # Enable inline plots for interactivity
        plt.rcParams['figure.dpi'] = 150

        # Ignore warnings for cleaner output
        import warnings
        warnings.filterwarnings('ignore')

        print("Imports loaded successfully!")

        # Test that we can import our utilities
        from utils.preprocessing import *

        # Check that data directory exists
        import os
        print("Data directory exists:", os.path.exists('./data'))
```

```
Imports loaded successfully!
Data directory exists: True
```

Load in data

6/70, +0

Run your function to load in the preprocessed STL-10 data in the following split:

- 6000 training samples
- 1000 validation samples
- 1000 test samples
- 70 dimensional feature vector

```
In [2]: # Load the STL-10 dataset with our preprocessing pipeline
        # This will cache the processed features for faster subsequent runs
        X_train, y_train, X_val, y_val, X_test, y_test = load_stl10_data(
            n_train=6000,
            n_val=1000,
            n_test=1000,
            n_features=70
        )

        print("Shape of X_train: {}".format(X_train.shape))
        print("Shape of y_train: {}".format(y_train.shape))

        print(f"\n{X_train.shape[0]} training samples")
        print(f"{X_val.shape[0]} validation samples")
        print(f"{X_test.shape[0]} test samples")
        print(f"{X_train.shape[1]} features")
```

```
Shape of X_train: (6000, 70)
Shape of y_train: (6000,)

6000 training samples
1000 validation samples
1000 test samples
70 features
```

9/20th in sequence

The data is unsupervised because a unsolved binary classification dataset that is useful for debugging and visualizing what your MLP is learning. Points with $y_i = 0$ correspond inside a circle that lies within a larger cube of 1. Points with corresponding outside the non-linear class $y_i = 1$. Training on this CIFA dataset allows us to see how well our implementation of the MLP is performing on a non-trivial task that is solvable by a human.

Hints

- Bootstrap the CIFA dataset that our first well-tested is train if the MLP code `fit_binary_classification()` and then `fit_multi_class()` after we bumps arrives.
- Plot the decision boundary of your classifier. Use `plot_decision_boundary_binary()` and `plot_decision_boundary()` to see what your model learns, and to debug errors.
- Make the support vector of your y_i predicting pure equal, otherwise you might see an artifact

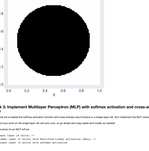
Deliverables

- Plot your model's error on different splits
- Plot a confusion matrix on the 1k validation
- Plot a confusion matrix on the 1k test set
- Plot a decision boundary on 2 of the features

```
In [3]: # Load binary data (2D)
        X_train_binary, y_train_binary, X_val_binary, y_val_binary, X_test_binary, y_test_binary = load_binary_dataset('./data', n_train=2000, n_val=500, n_test=500, n_features=2, random_state=42)

        print("Shape of X_train: {}".format(X_train_binary.shape))
        print("Shape of y_train: {}".format(y_train_binary.shape))

        print(f"\n{X_train_binary.shape[0]} training samples")
        print(f"{X_val_binary.shape[0]} validation samples")
        print(f"{X_test_binary.shape[0]} test samples")
        print(f"{X_train_binary.shape[1]} features")

        # Check class distribution
        unique_train, counts_train = np.unique(y_train_binary, return_counts=True)
        print("\nClass distribution in training set:")
        print(f"Class 0: {counts_train[0]} samples")
        print(f"Class 1: {counts_train[1]} samples")
```

```
Shape of X_train: (2000, 2)
Shape of y_train: (2000,)

2000 training samples
500 validation samples
500 test samples
2 features

Class distribution in training set:
Class 0: 1000 samples
Class 1: 1000 samples
```

```
In [4]: # Visualize the binary classification dataset
        fig, axes = plt.subplots(1, 3, figure=figure, dpi=150)

        # Training set
        scatter = axes[0].scatter(X_train_binary[:, 0], X_train_binary[:, 1], c=y_train_binary, cmap='bwr', alpha=0.6, s=20, edgecolors='k', linewidths=0.5)
        axes[0].set_title('Training Set')
        axes[0].set_xlabel('Feature 1')
        axes[0].set_ylabel('Feature 2')
        axes[0].grid(True, alpha=0.3)

        # Validation set
        scatter = axes[1].scatter(X_val_binary[:, 0], X_val_binary[:, 1], c=y_val_binary, cmap='bwr', alpha=0.6, s=20, edgecolors='k', linewidths=0.5)
        axes[1].set_title('Validation Set')
        axes[1].set_xlabel('Feature 1')
        axes[1].grid(True, alpha=0.3)

        # Test set
        scatter = axes[2].scatter(X_test_binary[:, 0], X_test_binary[:, 1], c=y_test_binary, cmap='bwr', alpha=0.6, s=20, edgecolors='k', linewidths=0.5)
        axes[2].set_title('Test Set')
        axes[2].set_xlabel('Feature 1')
        axes[2].grid(True, alpha=0.3)

        plt.tight_layout()
```



Task 3: Implement Multilayer Perceptron (MLP) with softmax activation and cross-entropy loss

Now that we understand the artificial neuron function and cross-entropy, have learned a logic layer, we will implement the MLP version.

Most of your work on this single layer will carry over, so go ahead and copy paste and modify as needed.

The structure of an MLP will be:

```
Input layer (2 features)
  → Hidden layer (n neurons) with ReLU/Sigmoid activation (Relu) →
  → Output layer (2 outputs) with softmax activation
```

3a. Implement the following functions in `mlp.py`

- `initialize_mlp`
- `sigmoid`
- `relu`
- `softmax`
- `mlp_forward`
- `mlp_backward`
- `fit_mlp`

3b. Test key functions with randomly generated data

```
In [5]: %run mlp.py
        # Create a simple MLP for debugging
        layer_sizes = [2, 4, 2]
        params = initialize_mlp(layer_sizes)

        print("Initialized parameters:")
        for key in sorted(params.keys):
            print(f"{key}: shape {params[key].shape}")
```

```
Initialized parameters:
b1: shape (4,)
b2: shape (2,)
W1: shape (2, 4)
W2: shape (4, 2)
```

```
In [6]: # Test activation functions and their derivatives
        x = np.array([-2, -1, 0, 1, 2])

        print("Testing sigmoid:")
        print("sigmoid(x):", sigmoid(x))
        print("sigmoid_derivative(x):", sigmoid_derivative(x))

        print("\nTesting relu:")
        print("relu(x):", relu(x))
        print("relu_derivative(x):", relu_derivative(x))

        print("\nTesting softmax:")
        logits = np.array([[1.0, 2.0, 3.0], [0.5, 1.5, 2.5]])
        print("softmax(logits):\n", softmax(logits))
        print("Row sums (should be 1.0):", softmax(logits).sum(axis=1))
```

```
Testing sigmoid:
sigmoid(x): [0.11920292 0.26894142 0.5        0.73105858 0.88079708]
sigmoid_derivative(x): [0.10499359 0.19661193 0.25       0.19661193 0.10499359]

Testing relu:
relu(x): [0 0 0 1 2]
relu_derivative(x): [0 0 0 1 1]

Testing softmax:
softmax(logits):
 [[0.09003057 0.24472847 0.66524096]
 [0.04742587 0.35206533 0.60050879]]
Row sums (should be 1.0): [1. 1.]
```

Test forward pass

```
In [7]: # Test forward pass with random data
        X = np.random.randn(3, 2)  # 3 samples, 2 features
        params = initialize_mlp([2, 4, 2])

        cache, loss = mlp_forward(X, params, y=None)

        print("Forward pass successful!")
        print("Loss:", loss)
        print("\nCache keys:", sorted(cache.keys))
        print("a1 (hidden activations) shape:", cache['a1'].shape)
        print("a2 (output probabilities) shape:", cache['a2'].shape)
```

```
Forward pass successful!
Loss: None

Cache keys: ['a1', 'a2', 'b1', 'b2', 'W1', 'W2', 'X', 'z1', 'z2']
a1 (hidden activations) shape: (3, 4)
a2 (output probabilities) shape: (3, 2)
```

Test loss

```
In [8]: # Test loss computation
        X = np.random.randn(3, 2)
        y = np.array([0, 1, 0])
        params = initialize_mlp([2, 4, 2])

        cache, loss = mlp_forward(X, params, y=y)

        print("Loss:", loss)
        print("\nExpected: ~0.693 (ln(2) for random 2-class)")
```

```
Loss: 0.7000000000000001

Expected: ~0.693 (ln(2) for random 2-class)
```

Test backward pass

```
In [9]: # Test backward pass
        X = np.random.randn(3, 2)
        y = np.array([0, 1, 0])
        params = initialize_mlp([2, 4, 2])

        cache, loss = mlp_forward(X, params, y=y)
        grads = mlp_backward(cache, params)

        print("Backward pass successful!")
        print("\nGradient keys:", sorted(grads.keys))
        print("dW1 shape:", grads['dW1'].shape)
        print("dW2 shape:", grads['dW2'].shape)
        print("db1 shape:", grads['db1'].shape)
        print("db2 shape:", grads['db2'].shape)
```

```
Backward pass successful!

Gradient keys: ['dW1', 'dW2', 'db1', 'db2']
dW1 shape: (2, 4)
dW2 shape: (4, 2)
db1 shape: (4,)
db2 shape: (2,)
```

Test gradient check

```
In [10]: # Test gradient check (this should pass!)
         X = np.random.randn(3, 2)
         y = np.array([0, 1, 0])
         params = initialize_mlp([2, 4, 2])

         cache, loss = mlp_forward(X, params, y=y)
         grads = mlp_backward(cache, params)

         print("Running gradient check...")
         print("This compares analytical gradients to numerical gradients")
         print("\nIf implementation is correct, relative errors should be < 1e-7")
         print("If you see errors > 1e-5, there's a bug in your backward pass")
```

```
Running gradient check...
This compares analytical gradients to numerical gradients

If implementation is correct, relative errors should be < 1e-7
If you see errors > 1e-5, there's a bug in your backward pass
```

Test full training

```
In [11]: # Train on binary classification
         params = initialize_mlp([2, 4, 2])

         params, losses = fit_mlp(X_train_binary, y_train_binary,
                          params=params,
                          learning_rate=0.1,
                          batch_size=32,
                          n_epochs=100,
                          verbose=True,
                          print_every=10)

         print("\nFinal loss:", losses[-1])
```

```
Epoch    0, Loss: 0.6931
Epoch   10, Loss: 0.6931
Epoch   20, Loss: 0.6931
Epoch   30, Loss: 0.6931
Epoch   40, Loss: 0.6931
Epoch   50, Loss: 0.6931
Epoch   60, Loss: 0.6931
Epoch   70, Loss: 0.6931
Epoch   80, Loss: 0.6931
Epoch   90, Loss: 0.6931

Final loss: 0.6931471805599453
```

Test predictions

```
In [12]: # Make predictions
         y_pred = predict(X_train_binary, params)
         y_prob = predict_proba(X_train_binary, params)

         print("Predictions shape:", y_pred.shape)
         print("Probabilities shape:", y_prob.shape)
         print("\nFirst 10 predictions:", y_pred[:10])
         print("First 10 probabilities:\n", y_prob[:10])
```

```
Predictions shape: (2000,)
Probabilities shape: (2000, 2)

First 10 predictions: [0 0 0 0 0 0 0 0 0 0]
First 10 probabilities:
 [[0.5 0.5]
 [0.5 0.5]
 [0.5 0.5]
 [0.5 0.5]
 [0.5 0.5]
 [0.5 0.5]
 [0.5 0.5]
 [0.5 0.5]
 [0.5 0.5]
 [0.5 0.5]]
```

Test accuracy

```
In [13]: # Calculate accuracy
         accuracy = np.mean(y_pred == y_train_binary)
         print("Accuracy:", accuracy)
         print("\nExpected: ~0.5 (random guessing)")
```

```
Accuracy: 0.5

Expected: ~0.5 (random guessing)
```

Test full training with more epochs

```
In [14]: # Train longer
         params = initialize_mlp([2, 4, 2])

         params, losses = fit_mlp(X_train_binary, y_train_binary,
                          params=params,
                          learning_rate=0.1,
                          batch_size=32,
                          n_epochs=1000,
                          verbose=True,
                          print_every=100)

         print("\nFinal loss:", losses[-1])
         print("Accuracy:", np.mean(predict(X_train_binary, params) == y_train_binary))
```

```
Epoch    0, Loss: 0.6931
Epoch  100, Loss: 0.6931
Epoch  200, Loss: 0.6931
Epoch  300, Loss: 0.6931
Epoch  400, Loss: 0.6931
Epoch  500, Loss: 0.6931
Epoch  600, Loss: 0.6931
Epoch  700, Loss: 0.6931
Epoch  800, Loss: 0.6931
Epoch  900, Loss: 0.6931

Final loss: 0.6931471805599453
Accuracy: 0.5
```

Test decision boundary

```
In [15]: # Visualize decision boundary
         plot_decision_boundary_binary(params, X_train_binary, y_train_binary,
                                       title='Untrained MLP')
```

Test learning curves

```
In [16]: # Plot learning curves
         plot_learning_curves(losses, title='Training Loss')
```

Test full workflow

```
In [17]: # Complete workflow
         params = initialize_mlp([2, 4, 2])

         params, losses = fit_mlp(X_train_binary, y_train_binary,
                          params=params,
                          learning_rate=0.1,
                          batch_size=32,
                          n_epochs=500,
                          verbose=True,
                          print_every=50)

         y_pred = predict(X_train_binary, params)
         accuracy = np.mean(y_pred == y_train_binary)
         print("\nFinal accuracy:", accuracy)
         print("\nIf this is ~0.5, something is wrong")
         print("If this is > 0.6, the network is learning")
```

```
Epoch    0, Loss: 0.6931
Epoch   50, Loss: 0.6931
Epoch  100, Loss: 0.6931
Epoch  150, Loss: 0.6931
Epoch  200, Loss: 0.6931
Epoch  250, Loss: 0.6931
Epoch  300, Loss: 0.6931
Epoch  350, Loss: 0.6931
Epoch  400, Loss: 0.6931
Epoch  450, Loss: 0.6931

Final accuracy: 0.5

If this is ~0.5, something is wrong
If this is > 0.6, the network is learning
```

Test hidden layer size

```
In [18]: # Compare different hidden layer sizes
         for hidden_size in [2, 4, 8, 16]:
             params = initialize_mlp([2, hidden_size, 2])
             params, losses = fit_mlp(X_train_binary, y_train_binary,
                              params=params,
                              learning_rate=0.1,
                              batch_size=32,
                              n_epochs=500,
                              verbose=False)
             y_pred = predict(X_train_binary, params)
             accuracy = np.mean(y_pred == y_train_binary)
             print(f"Hidden size {hidden_size}: Final loss = {losses[-1]:.4f}, Accuracy = {accuracy}")
```

```
Hidden size 2: Final loss = 0.6931, Accuracy = 0.5
Hidden size 4: Final loss = 0.6931, Accuracy = 0.5
Hidden size 8: Final loss = 0.6931, Accuracy = 0.5
Hidden size 16: Final loss = 0.6931, Accuracy = 0.5
```

Test learning rate

```
In [19]: # Compare different learning rates
         for lr in [0.01, 0.1, 1.0]:
             params = initialize_mlp([2, 4, 2])
             params, losses = fit_mlp(X_train_binary, y_train_binary,
                              params=params,
                              learning_rate=lr,
                              batch_size=32,
                              n_epochs=500,
                              verbose=False)
             y_pred = predict(X_train_binary, params)
             accuracy = np.mean(y_pred == y_train_binary)
             print(f"Learning rate {lr}: Final loss = {losses[-1]:.4f}, Accuracy = {accuracy}")
```

```
Learning rate 0.01: Final loss = 0.6931, Accuracy = 0.5
Learning rate 0.1: Final loss = 0.6931, Accuracy = 0.5
Learning rate 1.0: Final loss = 0.6931, Accuracy = 0.5
```

Test batch size

```
In [20]: # Compare different batch sizes
         for batch_size in [8, 32, 128, 512]:
             params = initialize_mlp([2, 4, 2])
             params, losses = fit_mlp(X_train_binary, y_train_binary,
                              params=params,
                              learning_rate=0.1,
                              batch_size=batch_size,
                              n_epochs=500,
                              verbose=False)
             y_pred = predict(X_train_binary, params)
             accuracy = np.mean(y_pred == y_train_binary)
             print(f"Batch size {batch_size}: Final loss = {losses[-1]:.4f}, Accuracy = {accuracy}")
```

```
Batch size 8: Final loss = 0.6931, Accuracy = 0.5
Batch size 32: Final loss = 0.6931, Accuracy = 0.5
Batch size 128: Final loss = 0.6931, Accuracy = 0.5
Batch size 512: Final loss = 0.6931, Accuracy = 0.5
```

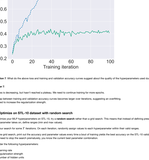
Test full workflow

```
In [21]: # Complete workflow
         params = initialize_mlp([2, 4, 2])

         params, losses = fit_mlp(X_train_binary, y_train_binary,
                          params=params,
                          learning_rate=0.1,
                          batch_size=32,
                          n_epochs=500,
                          verbose=True,
                          print_every=50)

         y_pred = predict(X_train_binary, params)
         accuracy = np.mean(y_pred == y_train_binary)
         print("\nFinal accuracy:", accuracy)
         print("\nIf this is ~0.5, something is wrong")
         print("If this is > 0.6, the network is learning")
```

```
Epoch    0, Loss: 0.6931
Epoch   50, Loss: 0.6931
Epoch  100, Loss: 0.6931
Epoch  150, Loss: 0.6931
Epoch  200, Loss: 0.6931
Epoch  250, Loss: 0.6931
Epoch  300, Loss: 0.6931
Epoch  350, Loss: 0.6931
Epoch  400, Loss: 0.6931
Epoch  450, Loss: 0.6931

Final accuracy: 0.5

If this is ~0.5, something is wrong
If this is > 0.6, the network is learning
```

Test decision boundary

```
In [22]: # Visualize decision boundary
         plot_decision_boundary_binary(params, X_train_binary, y_train_binary,
                                       title='Trained MLP')
```



Test learning curves

```
In [23]: # Plot learning curves
         plot_learning_curves(losses, title='Training Loss')
```


Test full workflow

```
In [24]: # Complete workflow
         params = initialize_mlp([2, 4, 2])

         params, losses = fit_mlp(X_train_binary, y_train_binary,
                          params=params,
                          learning_rate=0.1,
                          batch_size=32,
                          n_epochs=500,
                          verbose=True,
                          print_every=50)

         y_pred = predict(X_train_binary, params)
         accuracy = np.mean(y_pred == y_train_binary)
         print("\nFinal accuracy:", accuracy)
         print("\nIf this is ~0.5, something is wrong")
         print("If this is > 0.6, the network is learning")
```

```
Epoch    0, Loss: 0.6931
Epoch   50, Loss: 0.6931
Epoch  100, Loss: 0.6931
Epoch  150, Loss: 0.6931
Epoch  200, Loss: 0.6931
Epoch  250, Loss: 0.6931
Epoch  300, Loss: 0.6931
Epoch  350, Loss: 0.6931
Epoch  400, Loss: 0.6931
Epoch  450, Loss: 0.6931

Final accuracy: 0.5

If this is ~0.5, something is wrong
If this is > 0.6, the network is learning
```

Test decision boundary

```
In [25]: # Visualize decision boundary
         plot_decision_boundary_binary(params, X_train_binary, y_train_binary,
                                       title='Trained MLP')
```


Test learning curves

```
In [26]: # Plot learning curves
         plot_learning_curves(losses, title='Training Loss')
```


Test full workflow

```
In [27]: # Complete workflow
         params = initialize_mlp([2, 4, 2])

         params, losses = fit_mlp(X_train_binary, y_train_binary,
                          params=params,
                          learning_rate=0.1,
                          batch_size=32,
                          n_epochs=500,
                          verbose=True,
                          print_every=50)

         y_pred = predict(X_train_binary, params)
         accuracy = np.mean(y_pred == y_train_binary)
         print("\nFinal accuracy:", accuracy)
         print("\nIf this is ~0.5, something is wrong")
         print("If this is > 0.6, the network is learning")
```

```
Epoch    0, Loss: 0.6931
Epoch   50, Loss: 0.6931
Epoch  100, Loss: 0.6931
Epoch  150, Loss: 0.6931
Epoch  200, Loss: 0.6931
Epoch  250, Loss: 0.6931
Epoch  300, Loss: 0.6931
Epoch  350, Loss: 0.6931
Epoch  400, Loss: 0.6931
Epoch  450, Loss: 0.6931

Final accuracy: 0.5

If this is ~0.5, something is wrong
If this is > 0.6, the network is learning
```

Test decision boundary

```
In [28]: # Visualize decision boundary
         plot_decision_boundary_binary(params, X_train_binary, y_train_binary,
                                       title='Trained MLP')
```


Test learning curves

```
In [29]: # Plot learning curves
         plot_learning_curves(losses, title='Training Loss')
```


Test full workflow

```
In [30]: # Complete workflow
         params = initialize_mlp([2, 4, 2])

         params, losses = fit_mlp(X_train_binary, y_train_binary,
                          params=params,
                          learning_rate=0.1,
                          batch_size=32,
                          n_epochs=500,
                          verbose=True,
                          print_every=50)

         y_pred = predict(X_train_binary, params)
         accuracy = np.mean(y_pred == y_train_binary)
         print("\nFinal accuracy:", accuracy)
         print("\nIf this is ~0.5, something is wrong")
         print("If this is > 0.6, the network is learning")
```

```
Epoch    0, Loss: 0.6931
Epoch   50, Loss: 0.6931
Epoch  100, Loss: 0.6931
Epoch  150, Loss: 0.6931
Epoch  200, Loss: 0.6931
Epoch  250, Loss: 0.6931
Epoch  300, Loss: 0.6931
Epoch  350, Loss: 0.6931
Epoch  400, Loss: 0.6931
Epoch  450, Loss: 0.6931

Final accuracy: 0.5

If this is ~0.5, something is wrong
If this is > 0.6, the network is learning
```

Test decision boundary

```
In [31]: # Visualize decision boundary
         plot_decision_boundary_binary(params, X_train_binary, y_train_binary,
                                       title='Trained MLP')
```


Test learning curves

```
In [32]: # Plot learning curves
         plot_learning_curves(losses, title='Training Loss')
```


Test full workflow

```
In [33]: # Complete workflow
         params = initialize_mlp([2, 4, 2])

         params, losses = fit_mlp(X_train_binary, y_train_binary,
                          params=params,
                          learning_rate=0.1,
                          batch_size=32,
                          n_epochs=500,
                          verbose=True,
                          print_every=50)

         y_pred = predict(X_train_binary, params)
         accuracy = np.mean(y_pred == y_train_binary)
         print("\nFinal accuracy:", accuracy)
         print("\nIf this is ~0.5, something is wrong")
         print("If this is > 0.6, the network is learning")
```

```
Epoch    0, Loss: 0.6931
Epoch   50, Loss: 0.6931
Epoch  100, Loss: 0.6931
Epoch  150, Loss: 0.6931
Epoch  200, Loss: 0.6931
Epoch  250, Loss: 0.6931
Epoch  300, Loss: 0.6931
Epoch  350, Loss: 0.6931
Epoch  400, Loss: 0.6931
Epoch  450, Loss: 0.6931

Final accuracy: 0.5

If this is ~0.5, something is wrong
If this is > 0.6, the network is learning
```

Test decision boundary

```
In [34]: # Visualize decision boundary
         plot_decision_boundary_binary(params, X_train_binary, y_train_binary,
                                       title='Trained MLP')
```


Test learning curves

```
In [35]: # Plot learning curves
         plot_learning_curves(losses, title='Training Loss')
```


Test full workflow

```
In [36]: # Complete workflow
         params = initialize_mlp([2, 4, 2])

         params, losses = fit_mlp(X_train_binary, y_train_binary,
                          params=params,
                          learning_rate=0.1,
                          batch_size=32,
                          n_epochs=500,
                          verbose=True,
                          print_every=50)

         y_pred = predict(X_train_binary, params)
         accuracy = np.mean(y_pred == y_train_binary)
         print("\nFinal accuracy:", accuracy)
         print("\nIf this is ~0.5, something is wrong")
         print("If this is > 0.6, the network is learning")
```

```
Epoch    0, Loss: 0.6931
Epoch   50, Loss: 0.6931
Epoch  100, Loss: 0.6931
Epoch  150, Loss: 0.6931
Epoch  200, Loss: 0.6931
Epoch  250, Loss: 0.6931
Epoch  300, Loss: 0.6931
Epoch  350, Loss: 0.6931
Epoch  400, Loss: 0.6931
Epoch  450, Loss: 0.6931

Final accuracy: 0.5

If this is ~0.5, something is wrong
If this is > 0.6, the network is learning
```

Test decision boundary

```
In [37]: # Visualize decision boundary
         plot_decision_boundary_binary(params, X_train_binary, y_train_binary,
                                       title='Trained MLP')
```


Test learning curves

```
In [38]: # Plot learning curves
         plot_learning_curves(losses, title='Training Loss')
```


Test full workflow

```
In [39]: # Complete workflow
         params = initialize_mlp([2, 4, 2])

         params, losses = fit_mlp(X_train_binary, y_train_binary,
                          params=params,
                          learning_rate=0.1,
                          batch_size=32,
                          n_epochs=500,
                          verbose=True,
                          print_every=50)

         y_pred = predict(X_train_binary, params)
         accuracy = np.mean(y_pred == y_train_binary)
         print("\nFinal accuracy:", accuracy)
         print("\nIf this is ~0.5, something is wrong")
         print("If this is > 0.6, the network is learning")
```

```
Epoch    0, Loss: 0.6931
Epoch   50, Loss: 0.6931
Epoch  100, Loss: 0.6931
Epoch  150, Loss: 0.6931
Epoch  200, Loss: 0.6931
Epoch  250, Loss: 0.6931
Epoch  300, Loss: 0.6931
Epoch  350, Loss: 0.6931
Epoch  400, Loss: 0.6931
Epoch  450, Loss: 0.6931

Final accuracy: 0.5

If this is ~0.5, something is wrong
If this is > 0.6, the network is learning
```

Test decision boundary

```
In [40]: # Visualize decision boundary
         plot_decision_boundary_binary(params, X_train_binary, y_train_binary,
                                       title='Trained MLP')
```


Test learning curves

```
In [41]: # Plot learning curves
         plot_learning_curves(losses, title='Training Loss')
```


Test full workflow

```
In [42]: # Complete workflow
         params = initialize_mlp([2, 4, 2])

         params, losses = fit_mlp(X_train_binary, y_train_binary,
                          params=params,
                          learning_rate=0.1,
                          batch_size=32,
                          n_epochs=500,
                          verbose=True,
                          print_every=50)

         y_pred = predict(X_train_binary, params)
         accuracy = np.mean(y_pred == y_train_binary)
         print("\nFinal accuracy:", accuracy)
         print("\nIf this is ~0.5, something is wrong")
         print("If this is > 0.6, the network is learning")
```

```
Epoch    0, Loss: 0.6931
Epoch   50, Loss: 0.6931
Epoch  100, Loss: 0.6931
Epoch  150, Loss: 0.6931
Epoch  200, Loss: 0.6931
Epoch  250, Loss: 0.6931
Epoch  300, Loss: 0.6931
Epoch  350, Loss: 0.6931
Epoch  400, Loss: 0.6931
Epoch  450, Loss: 0.6931

Final accuracy: 0.5

If this is ~0.5, something is wrong
If this is > 0.6, the network is learning
```

Test decision boundary

```
In [43]: # Visualize decision boundary
         plot_decision_boundary_binary(params, X_train_binary, y_train_binary,
                                       title='Trained MLP')
```


Test learning curves

```
In [44]: # Plot learning curves
         plot_learning_curves(losses, title='Training Loss')
```


Test full workflow

```
In [45]: # Complete workflow
         params = initialize_mlp([2, 4, 2])

         params, losses = fit_mlp(X_train_binary, y_train_binary,
                          params=params,
                          learning_rate=0.1,
                          batch_size=32,
                          n_epochs=500,
                          verbose=True,
                          print_every=50)

         y_pred = predict(X_train_binary, params)
         accuracy = np.mean(y_pred == y_train_binary)
         print("\nFinal accuracy:", accuracy)
         print("\nIf this is ~0.5, something is wrong")
         print("If this is > 0.6, the network is learning")
```

```
Epoch    0, Loss: 0.6931
Epoch   50, Loss: 0.6931
Epoch  100, Loss: 0.6931
Epoch  150, Loss: 0.6931
Epoch  200, Loss: 0.6931
Epoch  250, Loss: 0.6931
Epoch  300, Loss: 0.6931
Epoch  350, Loss: 0.6931
Epoch  400, Loss: 0.6931
Epoch  450, Loss: 0.6931

Final accuracy: 0.5

If this is ~0.5, something is wrong
If this is > 0.6, the network is learning
```

Test decision boundary

```
In [46]: # Visualize decision boundary
         plot_decision_boundary_binary(params, X_train_binary, y_train_binary,
                                       title='Trained MLP')
```


Test learning curves

```
In [47]: # Plot learning curves
         plot_learning_curves(losses, title='Training Loss')
```


Test full workflow

```
In [48]: # Complete workflow
         params = initialize_mlp([2, 4, 2])

         params, losses = fit_mlp(X_train_binary, y_train_binary,
                          params=params,
                          learning_rate=0.1,
                          batch_size=32,
                          n_epochs=500,
                          verbose=True,
                          print_every=50)

         y_pred = predict(X_train_binary, params)
         accuracy = np.mean(y_pred == y_train_binary)
         print("\nFinal accuracy:", accuracy)
         print("\nIf this is ~0.5, something is wrong")
         print("If this is > 0.6, the network is learning")
```

```
Epoch    0, Loss: 0.6931
Epoch   50, Loss: 0.6931
Epoch  100, Loss: 0.6931
Epoch  150, Loss: 0.6931
Epoch  200, Loss: 0.6931
Epoch  250, Loss: 0.6931
Epoch  300, Loss: 0.6931
Epoch  350, Loss: 0.6931
Epoch  400, Loss: 0.6931
Epoch  450, Loss: 0.6931

Final accuracy: 0.5

If this is ~0.5, something is wrong
If this is > 0.6, the network is learning
```

Test decision boundary

```
In [49]: # Visualize decision boundary
         plot_decision_boundary_binary(params, X_train_binary, y_train_binary,
                                       title='Trained MLP')
```


Test learning curves

```
In [50]: # Plot learning curves
         plot_learning_curves(losses, title='Training Loss')
```


Test full workflow

```
In [51]: # Complete workflow
         params = initialize_mlp([2, 4, 2])

         params, losses = fit_mlp(X_train_binary, y_train_binary,
                          params=params,
                          learning_rate=0.1,
                          batch_size=32,
                          n_epochs=500,
                          verbose=True,
                          print_every=50)

         y_pred = predict(X_train_binary, params)
         accuracy = np.mean(y_pred == y_train_binary)
         print("\nFinal accuracy:", accuracy)
         print("\nIf this is ~0.5, something is wrong")
         print("If this is > 0.6, the network is learning")
```

```
Epoch    0, Loss: 0.6931
Epoch   50, Loss: 0.6931
Epoch  100, Loss: 0.6931
Epoch  150, Loss: 0.6931
Epoch  200, Loss: 0.6931
Epoch  250, Loss: 0.6931
Epoch  300, Loss: 0.6931
Epoch  350, Loss: 0.6931
Epoch  400, Loss: 0.6931
Epoch  450, Loss: 0.6931

Final accuracy: 0.5

If this is ~0.5, something is wrong
If this is > 0.6, the network is learning
```

Test decision boundary

```
In [52]: # Visualize decision boundary
         plot_decision_boundary_binary(params, X_train_binary, y_train_binary,
                                       title='Trained MLP')
```


Test learning curves

```
In [53]: # Plot learning curves
         plot_learning_curves(losses, title='Training Loss')
```


Test full workflow

```
In [54]: # Complete workflow
         params = initialize_mlp([2, 4, 2])

         params, losses = fit_mlp(X_train_binary, y_train_binary,
                          params=params,
                          learning_rate=0.1,
                          batch_size=32,
                          n_epochs=500,
                          verbose=True,
                          print_every=50)

         y_pred = predict(X_train_binary, params)
         accuracy = np.mean(y_pred == y_train_binary)
         print("\nFinal accuracy:", accuracy)
         print("\nIf this is ~0.5, something is wrong")
         print("If this is > 0.6, the network is learning")
```

```
Epoch    0, Loss: 0.6931
Epoch   50, Loss: 0.6931
Epoch  100, Loss: 0.6931
Epoch  150, Loss: 0.6931
Epoch  200, Loss: 0.6931
Epoch  250, Loss: 0.6931
Epoch  300, Loss: 0.6931
Epoch  350, Loss: 0.6931
Epoch  400, Loss: 0.6931
Epoch  450, Loss: 0.6931

Final accuracy: 0.5

If this is ~0.5, something is wrong
If this is > 0.6, the network is learning
```

Test decision boundary

```
In [55]: # Visualize decision boundary
         plot_decision_boundary_binary(params, X_train_binary, y_train_binary,
                                       title='Trained MLP')
```

In [2]:model = 1
layer = 1
neuron = 1
activation = 1
optimizer = 1
learning_rate = 1

```
def TUNING_MODEL(X_train, Y_train, X_val, Y_val, epochs, hidden_num, hidden_den, batch_size):
    global model
    global activation, optimizer, learning_rate
    model = Sequential()
    model.add(Dense(X_train.shape[1], input_dim=X_train.shape[1]))
    model.add(Activation('tanh'))

    for i in range(layer):
        model.add(Dense(int(hidden_num/hidden_den), init='lecun_uniform'))
        model.add(Activation('tanh'))
        model.add(Dropout(0.5))
    model.add(Dense(Y_train.shape[1], init='lecun_uniform'))
    model.add(Activation('softmax'))
    sgd = SGD(lr=learning_rate, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrac=['accuracy'])
    hist = model.fit(X_train, Y_train, nb_epoch=epochs, batch_size=batch_size,
                     verbose=0, validation_data=(X_val, Y_val))
    print("Finish Round of Random Search")
    print("Model Accuracy on training data: %s", hist.history['acc'][-1:])
    print("Model Accuracy on validation data: %s", hist.history['val_acc'][-1:])
    return model, hist.history['acc'], hist.history['val_acc'], hist.history['loss'], hist.history['val_loss']

TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)
```

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

In [3]:model, acc, val_acc, loss, val_loss = TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

Based on the first random research with very large range, the highest accuracy of training and validation is obtained during the 97th iterations with the hyperparameters:
hidden_num: 128
hidden_den: 2
epochs: 100
batch_size: 128
learning_rate: 1
activation: 1
optimizer: 1
Model Accuracy:0.99999999997

Based on the first random research, we obtained a decent 97% 'training' accuracy that we think the set of hyperparameters yields the best result so that it also has a decent value of validation accuracy. We will use the set of hyperparameters that yield the similar accuracy. There are still 5 of the range of the random search unused. We will use those in the next random search.

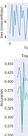
In [4]:model = 1
layer = 1
neuron = 1
activation = 1
optimizer = 1
learning_rate = 1

```
def TUNING_MODEL(X_train, Y_train, X_val, Y_val, epochs, hidden_num, hidden_den, batch_size):
    global model
    global activation, optimizer, learning_rate
    model = Sequential()
    model.add(Dense(X_train.shape[1], input_dim=X_train.shape[1]))
    model.add(Activation('tanh'))

    for i in range(layer):
        model.add(Dense(int(hidden_num/hidden_den), init='lecun_uniform'))
        model.add(Activation('tanh'))
        model.add(Dropout(0.5))
    model.add(Dense(Y_train.shape[1], init='lecun_uniform'))
    model.add(Activation('softmax'))
    sgd = SGD(lr=learning_rate, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrac=['accuracy'])
    hist = model.fit(X_train, Y_train, nb_epoch=epochs, batch_size=batch_size,
                     verbose=0, validation_data=(X_val, Y_val))
    print("Finish Round of Random Search")
    print("Model Accuracy on training data: %s", hist.history['acc'][-1:])
    print("Model Accuracy on validation data: %s", hist.history['val_acc'][-1:])
    return model, hist.history['acc'], hist.history['val_acc'], hist.history['loss'], hist.history['val_loss']

TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)
```

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

In [5]:model, acc, val_acc, loss, val_loss = TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```



The best set is generated in the 75th iteration with the following hyperparameters:
hidden_num: 128
hidden_den: 2
epochs: 100
batch_size: 128
learning_rate: 1
activation: 1
optimizer: 1
Model Accuracy:0.9

Thus, we can take the set of hyperparameters to use in tuning the number of hidden neurons in the hidden level around affect the performance of the MLP.

In [6]:model = 1
layer = 1
neuron = 1
activation = 1
optimizer = 1
learning_rate = 1

```
def TUNING_MODEL(X_train, Y_train, X_val, Y_val, epochs, hidden_num, hidden_den, batch_size):
    global model
    global activation, optimizer, learning_rate
    model = Sequential()
    model.add(Dense(X_train.shape[1], input_dim=X_train.shape[1]))
    model.add(Activation('tanh'))

    for i in range(layer):
        model.add(Dense(int(hidden_num/hidden_den), init='lecun_uniform'))
        model.add(Activation('tanh'))
        model.add(Dropout(0.5))
    model.add(Dense(Y_train.shape[1], init='lecun_uniform'))
    model.add(Activation('softmax'))
    sgd = SGD(lr=learning_rate, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrac=['accuracy'])
    hist = model.fit(X_train, Y_train, nb_epoch=epochs, batch_size=batch_size,
                     verbose=0, validation_data=(X_val, Y_val))
    print("Finish Round of Random Search")
    print("Model Accuracy on training data: %s", hist.history['acc'][-1:])
    print("Model Accuracy on validation data: %s", hist.history['val_acc'][-1:])
    return model, hist.history['acc'], hist.history['val_acc'], hist.history['loss'], hist.history['val_loss']

TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)
```

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

In [7]:model, acc, val_acc, loss, val_loss = TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

It looks like there is only a small difference in the accuracy of validation. Based on our previous observation that it seems that we have more hidden neurons the better it yields better performance. We have increased the value of the upper bound of the search to see what can we find.

In [8]:model = 1
layer = 1
neuron = 1
activation = 1
optimizer = 1
learning_rate = 1

```
def TUNING_MODEL(X_train, Y_train, X_val, Y_val, epochs, hidden_num, hidden_den, batch_size):
    global model
    global activation, optimizer, learning_rate
    model = Sequential()
    model.add(Dense(X_train.shape[1], input_dim=X_train.shape[1]))
    model.add(Activation('tanh'))

    for i in range(layer):
        model.add(Dense(int(hidden_num/hidden_den), init='lecun_uniform'))
        model.add(Activation('tanh'))
        model.add(Dropout(0.5))
    model.add(Dense(Y_train.shape[1], init='lecun_uniform'))
    model.add(Activation('softmax'))
    sgd = SGD(lr=learning_rate, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrac=['accuracy'])
    hist = model.fit(X_train, Y_train, nb_epoch=epochs, batch_size=batch_size,
                     verbose=0, validation_data=(X_val, Y_val))
    print("Finish Round of Random Search")
    print("Model Accuracy on training data: %s", hist.history['acc'][-1:])
    print("Model Accuracy on validation data: %s", hist.history['val_acc'][-1:])
    return model, hist.history['acc'], hist.history['val_acc'], hist.history['loss'], hist.history['val_loss']

TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)
```

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

In [9]:model, acc, val_acc, loss, val_loss = TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

It looks like that we have a small increase in the accuracy of validation. However, the price of that would be a really big mess. Therefore, I would be a first random search on the number of hidden neurons, which is from 200 to 250.

In [10]:model = 1
layer = 1
neuron = 1
activation = 1
optimizer = 1
learning_rate = 1

```
def TUNING_MODEL(X_train, Y_train, X_val, Y_val, epochs, hidden_num, hidden_den, batch_size):
    global model
    global activation, optimizer, learning_rate
    model = Sequential()
    model.add(Dense(X_train.shape[1], input_dim=X_train.shape[1]))
    model.add(Activation('tanh'))

    for i in range(layer):
        model.add(Dense(int(hidden_num/hidden_den), init='lecun_uniform'))
        model.add(Activation('tanh'))
        model.add(Dropout(0.5))
    model.add(Dense(Y_train.shape[1], init='lecun_uniform'))
    model.add(Activation('softmax'))
    sgd = SGD(lr=learning_rate, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrac=['accuracy'])
    hist = model.fit(X_train, Y_train, nb_epoch=epochs, batch_size=batch_size,
                     verbose=0, validation_data=(X_val, Y_val))
    print("Finish Round of Random Search")
    print("Model Accuracy on training data: %s", hist.history['acc'][-1:])
    print("Model Accuracy on validation data: %s", hist.history['val_acc'][-1:])
    return model, hist.history['acc'], hist.history['val_acc'], hist.history['loss'], hist.history['val_loss']

TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)
```

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

In [11]:model, acc, val_acc, loss, val_loss = TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

Based on all the result above, I choose the hyperparameters:
hidden_num: 250
hidden_den: 2
epochs: 100
batch_size: 128
learning_rate: 1
activation: 1
optimizer: 1
Model Accuracy:1.0

3b: First STL: 10 results with best hyperparameters

Step 3: WITH with the best hyperparameters that you found from your parameter search and create two plots:

- Training RTL 10 loss curve
- Training and validation RTL 10 accuracy curve

In [12]:model = 1
layer = 1
neuron = 1
activation = 1
optimizer = 1
learning_rate = 1

```
def TUNING_MODEL(X_train, Y_train, X_val, Y_val, epochs, hidden_num, hidden_den, batch_size):
    global model
    global activation, optimizer, learning_rate
    model = Sequential()
    model.add(Dense(X_train.shape[1], input_dim=X_train.shape[1]))
    model.add(Activation('tanh'))

    for i in range(layer):
        model.add(Dense(int(hidden_num/hidden_den), init='lecun_uniform'))
        model.add(Activation('tanh'))
        model.add(Dropout(0.5))
    model.add(Dense(Y_train.shape[1], init='lecun_uniform'))
    model.add(Activation('softmax'))
    sgd = SGD(lr=learning_rate, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrac=['accuracy'])
    hist = model.fit(X_train, Y_train, nb_epoch=epochs, batch_size=batch_size,
                     verbose=0, validation_data=(X_val, Y_val))
    print("Finish Round of Random Search")
    print("Model Accuracy on training data: %s", hist.history['acc'][-1:])
    print("Model Accuracy on validation data: %s", hist.history['val_acc'][-1:])
    return model, hist.history['acc'], hist.history['val_acc'], hist.history['loss'], hist.history['val_loss']

TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)
```

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

In [13]:model, acc, val_acc, loss, val_loss = TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

Based on all of the result above, I choose the hyperparameters:
hidden_num: 250
hidden_den: 2
epochs: 100
batch_size: 128
learning_rate: 1
activation: 1
optimizer: 1
Model Accuracy:1.0

3b: First STL: 10 results with best hyperparameters

Step 3: WITH with the best hyperparameters that you found from your parameter search and create two plots:

- Training RTL 10 loss curve
- Training and validation RTL 10 accuracy curve

In [14]:model = 1
layer = 1
neuron = 1
activation = 1
optimizer = 1
learning_rate = 1

```
def TUNING_MODEL(X_train, Y_train, X_val, Y_val, epochs, hidden_num, hidden_den, batch_size):
    global model
    global activation, optimizer, learning_rate
    model = Sequential()
    model.add(Dense(X_train.shape[1], input_dim=X_train.shape[1]))
    model.add(Activation('tanh'))

    for i in range(layer):
        model.add(Dense(int(hidden_num/hidden_den), init='lecun_uniform'))
        model.add(Activation('tanh'))
        model.add(Dropout(0.5))
    model.add(Dense(Y_train.shape[1], init='lecun_uniform'))
    model.add(Activation('softmax'))
    sgd = SGD(lr=learning_rate, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrac=['accuracy'])
    hist = model.fit(X_train, Y_train, nb_epoch=epochs, batch_size=batch_size,
                     verbose=0, validation_data=(X_val, Y_val))
    print("Finish Round of Random Search")
    print("Model Accuracy on training data: %s", hist.history['acc'][-1:])
    print("Model Accuracy on validation data: %s", hist.history['val_acc'][-1:])
    return model, hist.history['acc'], hist.history['val_acc'], hist.history['loss'], hist.history['val_loss']

TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)
```

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

In [15]:model, acc, val_acc, loss, val_loss = TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```


Based on all of the result above, I choose the hyperparameters:
hidden_num: 250
hidden_den: 2
epochs: 100
batch_size: 128
learning_rate: 1
activation: 1
optimizer: 1
Model Accuracy:1.0

3b: First STL: 10 results with best hyperparameters

Step 3: WITH with the best hyperparameters that you found from your parameter search and create two plots:

- Training RTL 10 loss curve
- Training and validation RTL 10 accuracy curve

In [16]:model = 1
layer = 1
neuron = 1
activation = 1
optimizer = 1
learning_rate = 1

```
def TUNING_MODEL(X_train, Y_train, X_val, Y_val, epochs, hidden_num, hidden_den, batch_size):
    global model
    global activation, optimizer, learning_rate
    model = Sequential()
    model.add(Dense(X_train.shape[1], input_dim=X_train.shape[1]))
    model.add(Activation('tanh'))

    for i in range(layer):
        model.add(Dense(int(hidden_num/hidden_den), init='lecun_uniform'))
        model.add(Activation('tanh'))
        model.add(Dropout(0.5))
    model.add(Dense(Y_train.shape[1], init='lecun_uniform'))
    model.add(Activation('softmax'))
    sgd = SGD(lr=learning_rate, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrac=['accuracy'])
    hist = model.fit(X_train, Y_train, nb_epoch=epochs, batch_size=batch_size,
                     verbose=0, validation_data=(X_val, Y_val))
    print("Finish Round of Random Search")
    print("Model Accuracy on training data: %s", hist.history['acc'][-1:])
    print("Model Accuracy on validation data: %s", hist.history['val_acc'][-1:])
    return model, hist.history['acc'], hist.history['val_acc'], hist.history['loss'], hist.history['val_loss']

TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)
```

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

In [17]:model, acc, val_acc, loss, val_loss = TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

Based on all of the result above, I choose the hyperparameters:
hidden_num: 250
hidden_den: 2
epochs: 100
batch_size: 128
learning_rate: 1
activation: 1
optimizer: 1
Model Accuracy:1.0

3b: First STL: 10 results with best hyperparameters

Step 3: WITH with the best hyperparameters that you found from your parameter search and create two plots:

- Training RTL 10 loss curve
- Training and validation RTL 10 accuracy curve

In [18]:model = 1
layer = 1
neuron = 1
activation = 1
optimizer = 1
learning_rate = 1

```
def TUNING_MODEL(X_train, Y_train, X_val, Y_val, epochs, hidden_num, hidden_den, batch_size):
    global model
    global activation, optimizer, learning_rate
    model = Sequential()
    model.add(Dense(X_train.shape[1], input_dim=X_train.shape[1]))
    model.add(Activation('tanh'))

    for i in range(layer):
        model.add(Dense(int(hidden_num/hidden_den), init='lecun_uniform'))
        model.add(Activation('tanh'))
        model.add(Dropout(0.5))
    model.add(Dense(Y_train.shape[1], init='lecun_uniform'))
    model.add(Activation('softmax'))
    sgd = SGD(lr=learning_rate, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrac=['accuracy'])
    hist = model.fit(X_train, Y_train, nb_epoch=epochs, batch_size=batch_size,
                     verbose=0, validation_data=(X_val, Y_val))
    print("Finish Round of Random Search")
    print("Model Accuracy on training data: %s", hist.history['acc'][-1:])
    print("Model Accuracy on validation data: %s", hist.history['val_acc'][-1:])
    return model, hist.history['acc'], hist.history['val_acc'], hist.history['loss'], hist.history['val_loss']

TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)
```

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

In [19]:model, acc, val_acc, loss, val_loss = TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

Based on all of the result above, I choose the hyperparameters:
hidden_num: 250
hidden_den: 2
epochs: 100
batch_size: 128
learning_rate: 1
activation: 1
optimizer: 1
Model Accuracy:1.0

3b: First STL: 10 results with best hyperparameters

Step 3: WITH with the best hyperparameters that you found from your parameter search and create two plots:

- Training RTL 10 loss curve
- Training and validation RTL 10 accuracy curve

In [20]:model = 1
layer = 1
neuron = 1
activation = 1
optimizer = 1
learning_rate = 1

```
def TUNING_MODEL(X_train, Y_train, X_val, Y_val, epochs, hidden_num, hidden_den, batch_size):
    global model
    global activation, optimizer, learning_rate
    model = Sequential()
    model.add(Dense(X_train.shape[1], input_dim=X_train.shape[1]))
    model.add(Activation('tanh'))

    for i in range(layer):
        model.add(Dense(int(hidden_num/hidden_den), init='lecun_uniform'))
        model.add(Activation('tanh'))
        model.add(Dropout(0.5))
    model.add(Dense(Y_train.shape[1], init='lecun_uniform'))
    model.add(Activation('softmax'))
    sgd = SGD(lr=learning_rate, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrac=['accuracy'])
    hist = model.fit(X_train, Y_train, nb_epoch=epochs, batch_size=batch_size,
                     verbose=0, validation_data=(X_val, Y_val))
    print("Finish Round of Random Search")
    print("Model Accuracy on training data: %s", hist.history['acc'][-1:])
    print("Model Accuracy on validation data: %s", hist.history['val_acc'][-1:])
    return model, hist.history['acc'], hist.history['val_acc'], hist.history['loss'], hist.history['val_loss']

TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)
```

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

In [21]:model, acc, val_acc, loss, val_loss = TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```


Based on all of the result above, I choose the hyperparameters:
hidden_num: 250
hidden_den: 2
epochs: 100
batch_size: 128
learning_rate: 1
activation: 1
optimizer: 1
Model Accuracy:1.0

3b: First STL: 10 results with best hyperparameters

Step 3: WITH with the best hyperparameters that you found from your parameter search and create two plots:

- Training RTL 10 loss curve
- Training and validation RTL 10 accuracy curve

In [22]:model = 1
layer = 1
neuron = 1
activation = 1
optimizer = 1
learning_rate = 1

```
def TUNING_MODEL(X_train, Y_train, X_val, Y_val, epochs, hidden_num, hidden_den, batch_size):
    global model
    global activation, optimizer, learning_rate
    model = Sequential()
    model.add(Dense(X_train.shape[1], input_dim=X_train.shape[1]))
    model.add(Activation('tanh'))

    for i in range(layer):
        model.add(Dense(int(hidden_num/hidden_den), init='lecun_uniform'))
        model.add(Activation('tanh'))
        model.add(Dropout(0.5))
    model.add(Dense(Y_train.shape[1], init='lecun_uniform'))
    model.add(Activation('softmax'))
    sgd = SGD(lr=learning_rate, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrac=['accuracy'])
    hist = model.fit(X_train, Y_train, nb_epoch=epochs, batch_size=batch_size,
                     verbose=0, validation_data=(X_val, Y_val))
    print("Finish Round of Random Search")
    print("Model Accuracy on training data: %s", hist.history['acc'][-1:])
    print("Model Accuracy on validation data: %s", hist.history['val_acc'][-1:])
    return model, hist.history['acc'], hist.history['val_acc'], hist.history['loss'], hist.history['val_loss']

TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)
```

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

In [23]:model, acc, val_acc, loss, val_loss = TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

Based on all of the result above, I choose the hyperparameters:
hidden_num: 250
hidden_den: 2
epochs: 100
batch_size: 128
learning_rate: 1
activation: 1
optimizer: 1
Model Accuracy:1.0

3b: First STL: 10 results with best hyperparameters

Step 3: WITH with the best hyperparameters that you found from your parameter search and create two plots:

- Training RTL 10 loss curve
- Training and validation RTL 10 accuracy curve

In [24]:model = 1
layer = 1
neuron = 1
activation = 1
optimizer = 1
learning_rate = 1

```
def TUNING_MODEL(X_train, Y_train, X_val, Y_val, epochs, hidden_num, hidden_den, batch_size):
    global model
    global activation, optimizer, learning_rate
    model = Sequential()
    model.add(Dense(X_train.shape[1], input_dim=X_train.shape[1]))
    model.add(Activation('tanh'))

    for i in range(layer):
        model.add(Dense(int(hidden_num/hidden_den), init='lecun_uniform'))
        model.add(Activation('tanh'))
        model.add(Dropout(0.5))
    model.add(Dense(Y_train.shape[1], init='lecun_uniform'))
    model.add(Activation('softmax'))
    sgd = SGD(lr=learning_rate, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrac=['accuracy'])
    hist = model.fit(X_train, Y_train, nb_epoch=epochs, batch_size=batch_size,
                     verbose=0, validation_data=(X_val, Y_val))
    print("Finish Round of Random Search")
    print("Model Accuracy on training data: %s", hist.history['acc'][-1:])
    print("Model Accuracy on validation data: %s", hist.history['val_acc'][-1:])
    return model, hist.history['acc'], hist.history['val_acc'], hist.history['loss'], hist.history['val_loss']

TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)
```

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

In [25]:model, acc, val_acc, loss, val_loss = TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

Based on all of the result above, I choose the hyperparameters:
hidden_num: 250
hidden_den: 2
epochs: 100
batch_size: 128
learning_rate: 1
activation: 1
optimizer: 1
Model Accuracy:1.0

3b: First STL: 10 results with best hyperparameters

Step 3: WITH with the best hyperparameters that you found from your parameter search and create two plots:

- Training RTL 10 loss curve
- Training and validation RTL 10 accuracy curve

In [26]:model = 1
layer = 1
neuron = 1
activation = 1
optimizer = 1
learning_rate = 1

```
def TUNING_MODEL(X_train, Y_train, X_val, Y_val, epochs, hidden_num, hidden_den, batch_size):
    global model
    global activation, optimizer, learning_rate
    model = Sequential()
    model.add(Dense(X_train.shape[1], input_dim=X_train.shape[1]))
    model.add(Activation('tanh'))

    for i in range(layer):
        model.add(Dense(int(hidden_num/hidden_den), init='lecun_uniform'))
        model.add(Activation('tanh'))
        model.add(Dropout(0.5))
    model.add(Dense(Y_train.shape[1], init='lecun_uniform'))
    model.add(Activation('softmax'))
    sgd = SGD(lr=learning_rate, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrac=['accuracy'])
    hist = model.fit(X_train, Y_train, nb_epoch=epochs, batch_size=batch_size,
                     verbose=0, validation_data=(X_val, Y_val))
    print("Finish Round of Random Search")
    print("Model Accuracy on training data: %s", hist.history['acc'][-1:])
    print("Model Accuracy on validation data: %s", hist.history['val_acc'][-1:])
    return model, hist.history['acc'], hist.history['val_acc'], hist.history['loss'], hist.history['val_loss']

TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)
```

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

In [27]:model, acc, val_acc, loss, val_loss = TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```


Based on all of the result above, I choose the hyperparameters:
hidden_num: 250
hidden_den: 2
epochs: 100
batch_size: 128
learning_rate: 1
activation: 1
optimizer: 1
Model Accuracy:1.0

3b: First STL: 10 results with best hyperparameters

Step 3: WITH with the best hyperparameters that you found from your parameter search and create two plots:

- Training RTL 10 loss curve
- Training and validation RTL 10 accuracy curve

In [28]:model = 1
layer = 1
neuron = 1
activation = 1
optimizer = 1
learning_rate = 1

```
def TUNING_MODEL(X_train, Y_train, X_val, Y_val, epochs, hidden_num, hidden_den, batch_size):
    global model
    global activation, optimizer, learning_rate
    model = Sequential()
    model.add(Dense(X_train.shape[1], input_dim=X_train.shape[1]))
    model.add(Activation('tanh'))

    for i in range(layer):
        model.add(Dense(int(hidden_num/hidden_den), init='lecun_uniform'))
        model.add(Activation('tanh'))
        model.add(Dropout(0.5))
    model.add(Dense(Y_train.shape[1], init='lecun_uniform'))
    model.add(Activation('softmax'))
    sgd = SGD(lr=learning_rate, decay=1e-6, momentum=0.9, nesterov=True)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrac=['accuracy'])
    hist = model.fit(X_train, Y_train, nb_epoch=epochs, batch_size=batch_size,
                     verbose=0, validation_data=(X_val, Y_val))
    print("Finish Round of Random Search")
    print("Model Accuracy on training data: %s", hist.history['acc'][-1:])
    print("Model Accuracy on validation data: %s", hist.history['val_acc'][-1:])
    return model, hist.history['acc'], hist.history['val_acc'], hist.history['loss'], hist.history['val_loss']

TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)
```

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```

In [29]:model, acc, val_acc, loss, val_loss = TUNING_MODEL(X_train, Y_train, X_val, Y_val, 100, 128, 2)

```
Finish Round of Random Search
Model Accuracy on training data: [u'0.9999999999999999']
Model Accuracy on validation data: [u'0.9999999999999999']
```


Based on all of the result above, I choose the hyperparameters:
hidden_num: 250
hidden_den: 2
epochs: 100
batch_size: 128
learning_rate: 1
activation: 1
optimizer: 1
Model Accuracy:1.0

3b: First STL: 10 results with best hyperparameters

Step 3: WITH with the best hyperparameters that you found from your parameter search and create two plots:

- Training R

```
In [1]: # Implement the accuracies of the train and
        # validation sets (accuracy_train, accuracy_val)
        accuracy_train, accuracy_val = 0.0, 0.0

        print("The accuracy of the train set: {}".accuracy_train)
        print("The accuracy of the test set: {}".accuracy_val)
```

```
The accuracy of the train set: 0.0
The accuracy of the test set: 0.0
```

Question 6: Use cross-fold-validation to compare the accuracy on the train set after 100 epochs. What accuracy do you get?

Answer 6: With cross-fold-validation and grid-search (after optimizing parameters as in question 7)

Answer 6: 99.0%, which is not as good.

Question 7: Implement a grid-search over hyperparameters in this case. It can be quite tedious. Find out how tensorflow's tf.contrib.learn.GridSearchCV can be used to automate this process. Find out which parameters are important to optimize (hint: the learning rate is one of them). Find out what range to search for each hyperparameter by looking the range of exploding.

Sy. Visualize learned weights

Run the code. Jupyter/ipyhon notebook is generally quite visualization of data.

```
In [2]: # Visualize the learned weights for each class.
        w = session.run(weights)
        fig, axes = plot_images(images=w.T,
                                cls=range(10),
                                ncols=5,
                                sharex=True,
                                sharey=True)
```


Extensions

Warning: Please do not integrate extensions into your base project so that it changes the expected behavior of your functions. It is better to duplicate the base project and add features from there.

Warning: Please do not use Keras or any other package wrapping with tensorflow! or, not every input variable is usually processed or an epoch's end sampling without replacement is a very, very, very,

n_epochs = 1000

With Replacement

time: 1.0338 s

loss: 0.0612
training accuracy: 1.0
validation accuracy: 0.9

Without Replacement

time elapsed: 0.8293 s

loss: 0.0496
training accuracy: 1.0
validation accuracy: 0.9

When n_epochs = 1000, sampling with replacement generated higher training loss and took more time than without replacement.

The traing and validation accuracies are the same for both methods.

However, in principle, sampling with replacement should be faster to compute.

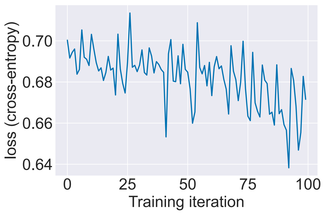
I am curious if this had something to do with the number of epochs.

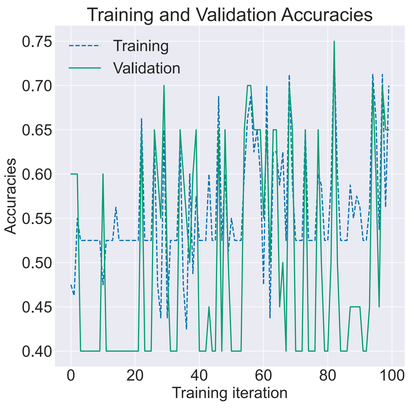
So I changed n_epochs to 100 to test the two methods.

With Replacement (n_epochs = 100)

In [80]:
```
cis_wr = MLP(2, 50, 2)

start = time.time()

loss_hist, acc_train, acc_valid = cis_wr.fit(cis_train_x, cis_train_y, cis_val_x, cis_val_y, reg=0, print_every=1000,
lr=0.5, mini_batch_sz=80, n_epochs=100, replace=True)

end = time.time()

print("\ntime elapsed:", end - start)

plot_cross_entropy_loss(loss_hist)

plot_train_valid_acc(acc_train, acc_valid)

print("loss:", loss_hist[-1], "\ntraining accuracy:", acc_train[-1], "\nvalidation accuracy:", acc_valid[-1])
```

```
Starting to train network...There will be 100 epochs and 100 iterations total, 1 iter/epoch.

    Completed Epoch 0/99.  Training loss: 0.70.    Validation Accuracy:60.0%

Finished training!

time elapsed: 0.2720329761505127
```



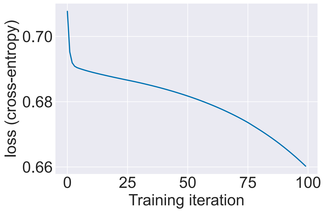


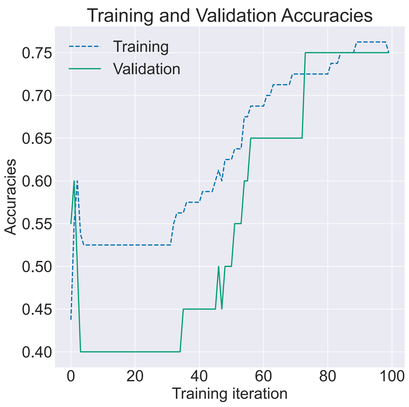

```
loss: 0.6715771113774673
training accuracy: 0.7
validation accuracy: 0.65
```

Without Replacement (n_epochs = 100)

In [83]:
```
cis_or = MLP(2, 50, 2)

start = time.time()

loss_hist, acc_train, acc_valid = cis_or.fit(cis_train_x, cis_train_y, cis_val_x, cis_val_y, reg=0, print_every=1000,
lr=0.5, mini_batch_sz=80, n_epochs=100, replace=False)

end = time.time()

print("\ntime elapsed:", end - start)

plot_cross_entropy_loss(loss_hist)

plot_train_valid_acc(acc_train, acc_valid)

print("loss:", loss_hist[-1], "\ntraining accuracy:", acc_train[-1], "\nvalidation accuracy:", acc_valid[-1])
```

```
Starting to train network...There will be 100 epochs and 100 iterations total, 1 iter/epoch.

    Completed Epoch 0/99.  Training loss: 0.71.    Validation Accuracy:55.0%

Finished training!

time elapsed: 0.23368000984191895
```






```
loss: 0.6602462097856365
training accuracy: 0.75
validation accuracy: 0.75
```

n_epochs = 100

With Replacement

time: 0.272 s

loss: 0.6716
training accuracy: 0.7
validation accuracy: 0.65

Without Replacement

time elapsed: 0.234 s

loss: 0.6602
training accuracy: 0.75
validation accuracy: 0.75

Sampling without replacement is still slightly faster. It generates higher training and validation accuracies than without replacement, but they do not differ much.

This makes sense because with replacement, it is possible to draw the same data sample for a large number of times in extreme cases, which affects the accuracy.

In [ ]: