two_layer_nn

February 10, 2024

0.1 This is the 2-layer neural network notebook for ECE C147/C247 Homework #3

Please follow the notebook linearly to implement a two layer neural network.

Please print out the notebook entirely when completed.

The goal of this notebook is to give you experience with training a two layer neural network.

[2]:
```
import random
import numpy as np
from utils.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%matplotlib inline
%load_ext autoreload
%autoreload 2

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

0.2 Toy example

Before loading CIFAR-10, there will be a toy example to test your implementation of the forward and backward pass. Make sure to read the description of TwoLayerNet class in neural_net.py file , understand the architecture and initializations

[5]:
```
from nndl.neural_net import TwoLayerNet
```

[6]:
```
# Create a small net and some toy data to check your implementations.
# Note that we set the random seed for repeatable experiments.

input_size = 4
hidden_size = 10
num_classes = 3
num_inputs = 5

def init_toy_model():
    np.random.seed(0)
```

```python
    return TwoLayerNet(input_size, hidden_size, num_classes, std=1e-1)

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

net = init_toy_model()
X, y = init_toy_data()
```

0.2.1 Compute forward pass scores

[30]:
```python
## Implement the forward pass of the neural network.
## See the loss() method in TwoLayerNet class for the same

# Note, there is a statement if y is None: return scores, which is why
# the following call will calculate the scores.
scores = net.loss(X)
print('Your scores:')
print(scores)
print()
print('correct scores:')
correct_scores = np.asarray([
    [-1.07260209,  0.05083871, -0.87253915],
    [-2.02778743, -0.10832494, -1.52641362],
    [-0.74225908,  0.15259725, -0.39578548],
    [-0.38172726,  0.10835902, -0.17328274],
    [-0.64417314, -0.18886813, -0.41106892]])
print(correct_scores)
print()

# The difference should be very small. We get < 1e-7
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))
```

```
Your scores:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
 [-0.38172726  0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

correct scores:
[[-1.07260209  0.05083871 -0.87253915]
 [-2.02778743 -0.10832494 -1.52641362]
 [-0.74225908  0.15259725 -0.39578548]
```

 [-0.38172726 0.10835902 -0.17328274]
 [-0.64417314 -0.18886813 -0.41106892]]

```
Difference between your scores and correct scores:
3.381231222787662e-08
```

0.2.2 Forward pass loss

[81]:
```python
loss, _ = net.loss(X, y, reg=0.05)
correct_loss = 1.071696123862817

# should be very small, we get < 1e-12
print("Loss:",loss)
print('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss)))
```

```
Loss: 1.071696123862817
Difference between your loss and correct loss:
0.0
```

0.2.3 Backward pass

Implements the backwards pass of the neural network. Check your gradients with the gradient check utilities provided.

[178]:
```python
from utils.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward␣
↪pass.
# If your implementation is correct, the difference between the numeric and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.

loss, grads = net.loss(X, y, reg=0.05)

# these should all be less than 1e-8 or so
for param_name in grads:
    f = lambda W: net.loss(X, y, reg=0.05)[0]
    param_grad_num = eval_numerical_gradient(f, net.params[param_name],␣
↪verbose=False)
    print('{} max relative error: {}'.format(param_name,␣
↪rel_error(param_grad_num, grads[param_name])))
#     print(param_grad_num)
```

```
b2 max relative error: 1.8392106647421603e-10
W2 max relative error: 3.4254767397498007e-10
b1 max relative error: 3.1726804786908923e-09
W1 max relative error: 1.2832908996874818e-09
```

0.2.4 Training the network

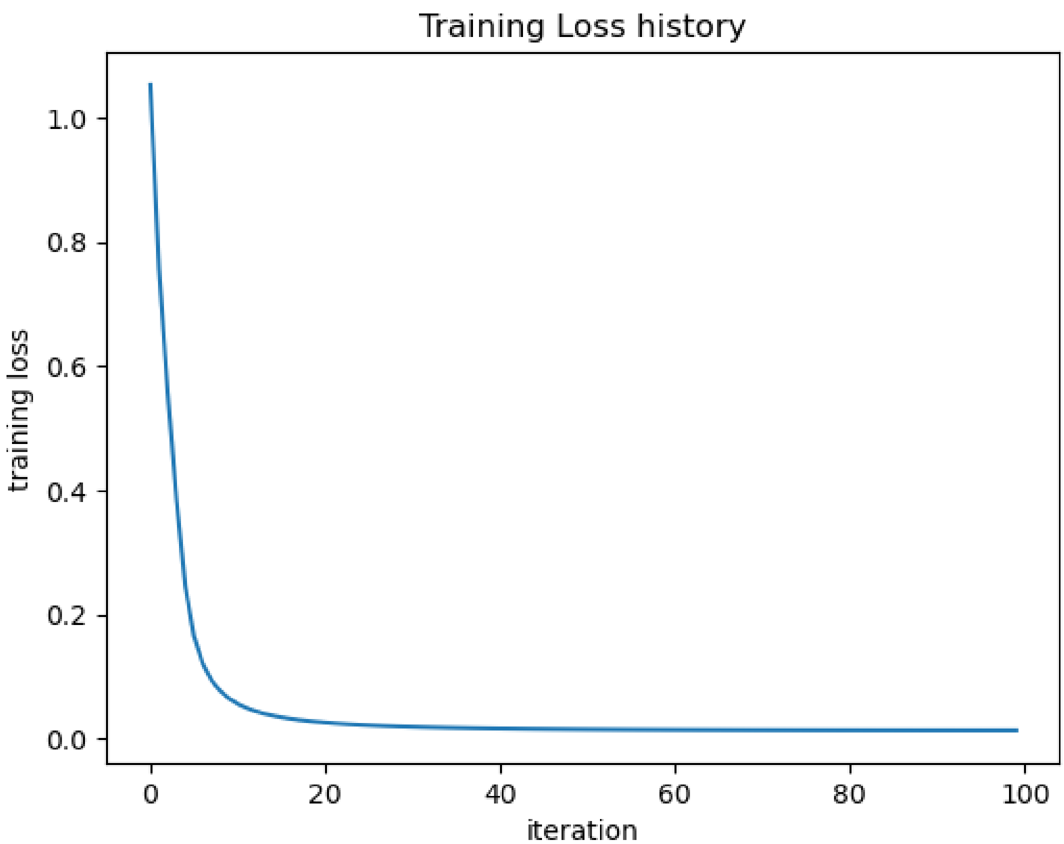
Implement neural_net.train() to train the network via stochastic gradient descent, much like the softmax and SVM.

[182]:
```
net = init_toy_model()
stats = net.train(X, y, X, y,
            learning_rate=1e-1, reg=5e-6,
            num_iters=100, verbose=False)

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

# plot the loss history
plt.plot(stats['loss_history'])
plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.show()
```

```
Final training loss:  0.013338037215396807
```



0.3 Classify CIFAR-10

Do classification on the CIFAR-10 dataset.

[184]:
```
from utils.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net classifier.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = '/home/andrea/git/UCLA/UCLA_ECE147/cifar-10-batches-py' #␣
↪remember to use correct path
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # Subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
    X_test = X_test.reshape(num_test, -1)

    return X_train, y_train, X_val, y_val, X_test, y_test


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
```

```python
print('Test labels shape: ', y_test.shape)
```

```
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3072)
Test labels shape:  (1000,)
```

0.3.1 Running SGD

If your implementation is correct, you should see a validation accuracy of around 28-29%.

[189]:
```python
input_size = 32 * 32 * 3
hidden_size = 50
num_classes = 10
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=1000, batch_size=200,
            learning_rate=1e-4, learning_rate_decay=0.95,
            reg=0.25, verbose=True)

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

# Save this net as the variable subopt_net for later comparison.
subopt_net = net
```

```
iteration 0 / 1000: loss 2.3027781109127994
iteration 100 / 1000: loss 2.3023051866249236
iteration 200 / 1000: loss 2.2971832381421646
iteration 300 / 1000: loss 2.2672203812398948
iteration 400 / 1000: loss 2.1834964063436906
iteration 500 / 1000: loss 2.162807507433956
iteration 600 / 1000: loss 2.077024585250963
iteration 700 / 1000: loss 1.9925966641437924
iteration 800 / 1000: loss 1.9871874501072693
iteration 900 / 1000: loss 1.9944860279554801
Validation accuracy:  0.28
```

0.4 Questions:

The training accuracy isn't great.

(1) What are some of the reasons why this is the case? Take the following cell to do some analyses and then report your answers in the cell following the one below.

(2) How should you fix the problems you identified in (1)?

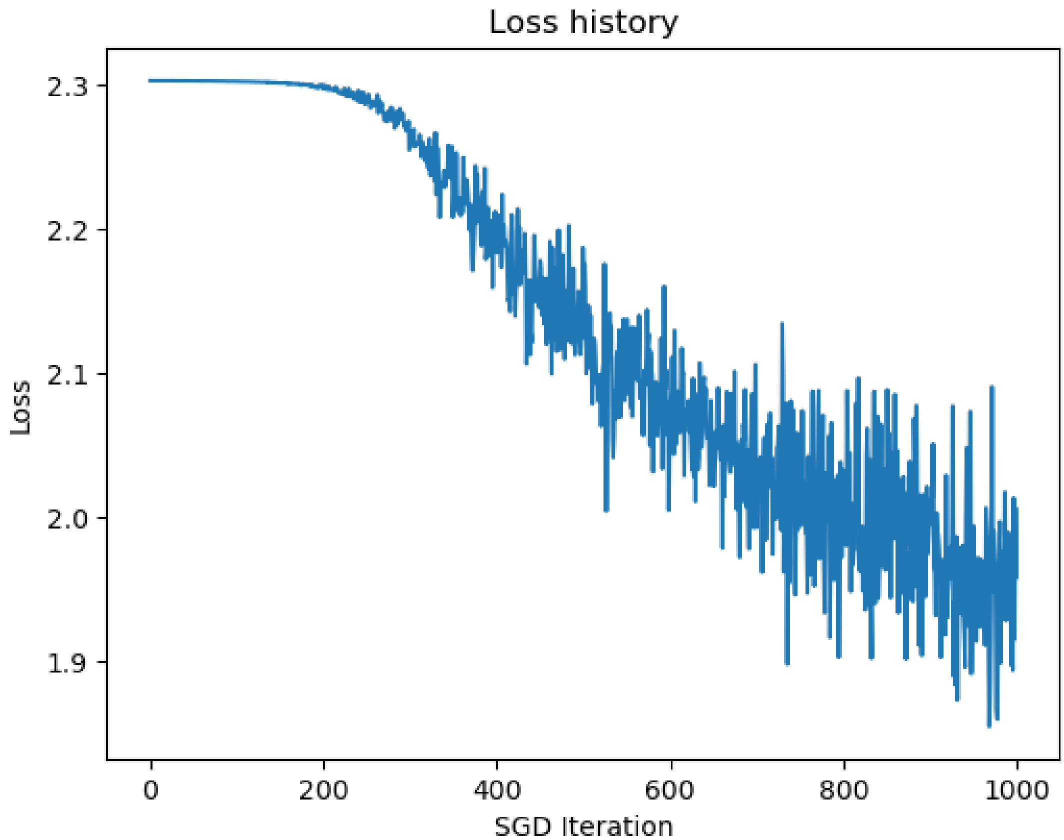
[190]:
```
stats['train_acc_history']
```

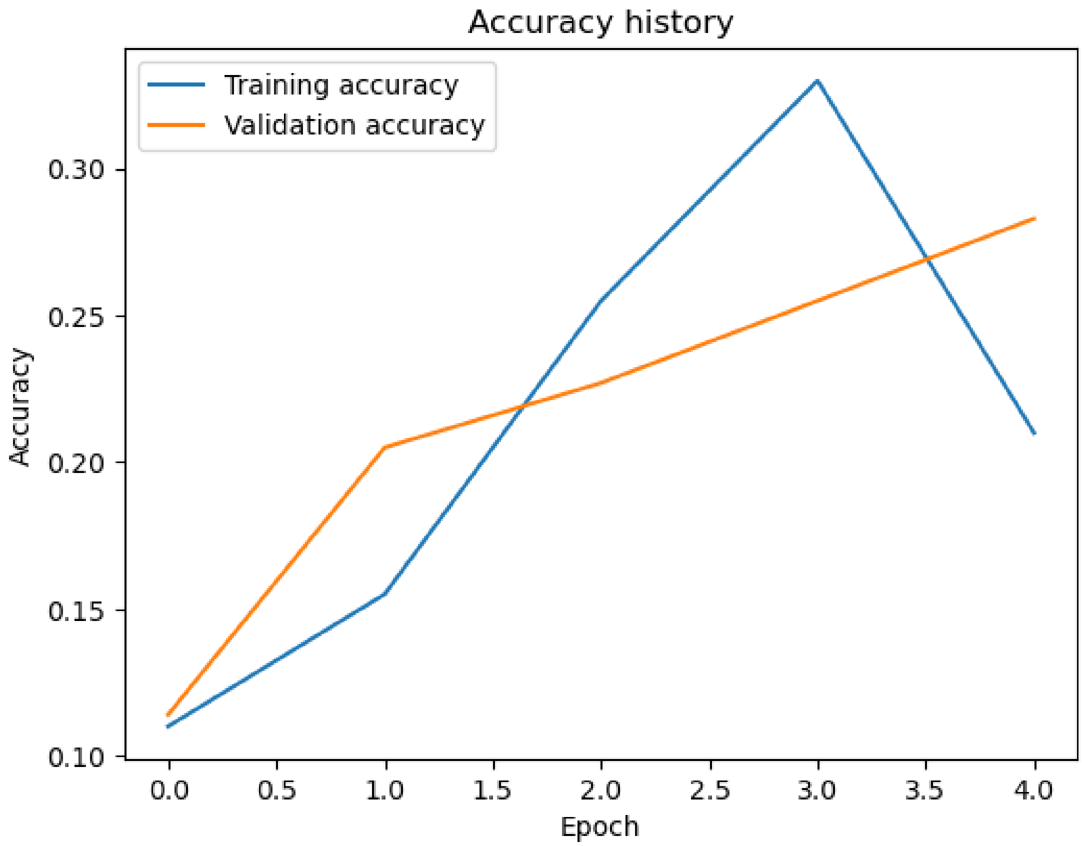
[190]:
```
[0.11, 0.155, 0.255, 0.33, 0.21]
```

[196]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Do some debugging to gain some insight into why the optimization
#   isn't great.
# ================================================================ #

# Plot the loss function and train / validation accuracies
plt.title("Loss history")
plt.plot(np.arange(len(stats['loss_history'])), stats['loss_history'])
plt.xlabel("SGD Iteration")
plt.ylabel("Loss")
plt.show()
plt.title("Accuracy history")
plt.plot(np.arange(len(stats['train_acc_history'])),␣
 ↪stats['train_acc_history'], label="Training accuracy")
plt.plot(np.arange(len(stats['val_acc_history'])), stats['val_acc_history'],␣
 ↪label="Validation accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.show()

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```


Loss history
2.3
2.2
2.1
2.0
1.9
Loss
0
200
400
600
800
1000
SGD Iteration



0.5 Answers:

(1) Towards the beginning of the loss history, loss stays roughly constant, indicating that the initial learning rate is too low. Interestingly, between epohs 3 and 4, training accuracy decreases as validation accuracy increases, the opposite of overfitting. Further, while loss is noisy, it does decrease somewhat linearly. This would suggest that the model could benefit from more training.

(2) I would change the hyperparameters of the network.

0.6 Optimize the neural network

Use the following part of the Jupyter notebook to optimize your hyperparameters on the validation set. Store your nets as best_net.

[233]:
```
best_net = None # store the best model into this

# ================================================================ #
# YOUR CODE HERE:
#   Optimize over your hyperparameters to arrive at the best neural
#   network.  You should be able to get over 50% validation accuracy.
```

```
#   For this part of the notebook, we will give credit based on the
#   accuracy you get.  Your score on this question will be multiplied by:
#      min(floor((X - 28%)) / %22, 1)
#   where if you get 50% or higher validation accuracy, you get full
#   points.
#
#   Note, you need to use the same network structure (keep hidden_size = 50)!
# ================================================================ #

best_pred = 0.0
# net = None

def nn_loss(args):
    global best_net, best_pred
    learning_rate, learning_rate_decay, reg = args

    net = TwoLayerNet(input_size, hidden_size, num_classes)

    # Train the network
    stats = net.train(X_train, y_train, X_val, y_val,
            num_iters=2250, batch_size=500,
            learning_rate=learning_rate,␣
 ↪learning_rate_decay=learning_rate_decay,
            reg=reg, verbose=False)
    val_acc = (net.predict(X_val) == y_val).mean()
    print('Iteration accuracy: ', val_acc)
    if (val_acc > best_pred):
        best_net = net
        best_pred = val_acc
    return -val_acc

result = opt.minimize(nn_loss, [5e-4,0.95,0.2],
                      method='Nelder-Mead',
                      tol=1e-4,
                      options={"maxiter": 10},
                      bounds=[(0,1),(0,1),(0,1)])

# best_net = TwoLayerNet(input_size, hidden_size, num_classes)

# # Train the network
# stats = best_net.train(X_train, y_train, X_val, y_val,
#         num_iters=2250, batch_size=500,
#         learning_rate=result["x"][0], learning_rate_decay=result["x"][1],
#         reg=result["x"][2], verbose=False)

# ================================================================ #
# END YOUR CODE HERE
```

```
# ================================================================ #
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

```
Iteration accuracy:  0.473
Iteration accuracy:  0.467
Iteration accuracy:  0.488
Iteration accuracy:  0.459
Iteration accuracy:  0.484
Iteration accuracy:  0.493
Iteration accuracy:  0.487
Iteration accuracy:  0.5
Iteration accuracy:  0.496
Iteration accuracy:  0.495
Iteration accuracy:  0.499
Iteration accuracy:  0.498
Iteration accuracy:  0.496
Iteration accuracy:  0.507
Iteration accuracy:  0.493
Iteration accuracy:  0.499
Iteration accuracy:  0.507
Validation accuracy:  0.507
```

[234]:
```
val_acc = (best_net.predict(X_val) == y_val).mean()
print('Validation accuracy: ', val_acc)
```

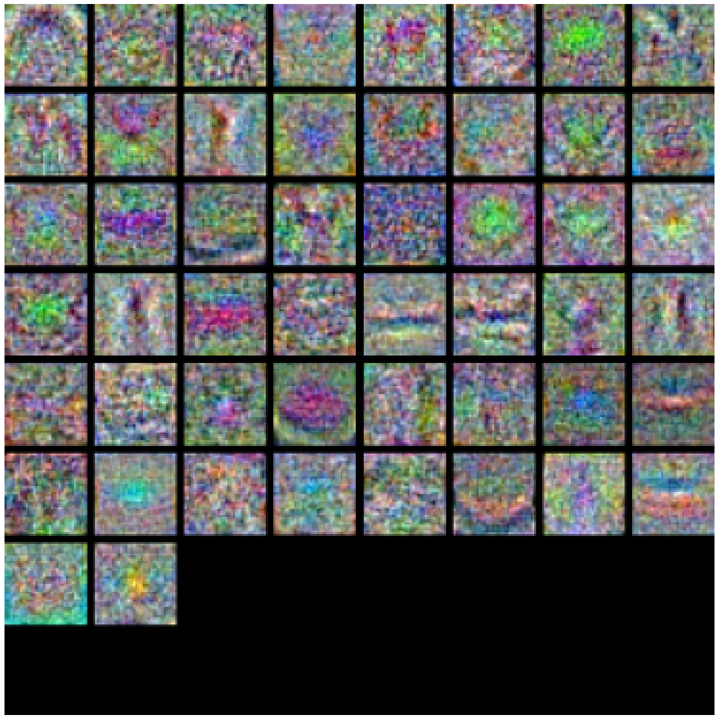
```
Validation accuracy:  0.507
```

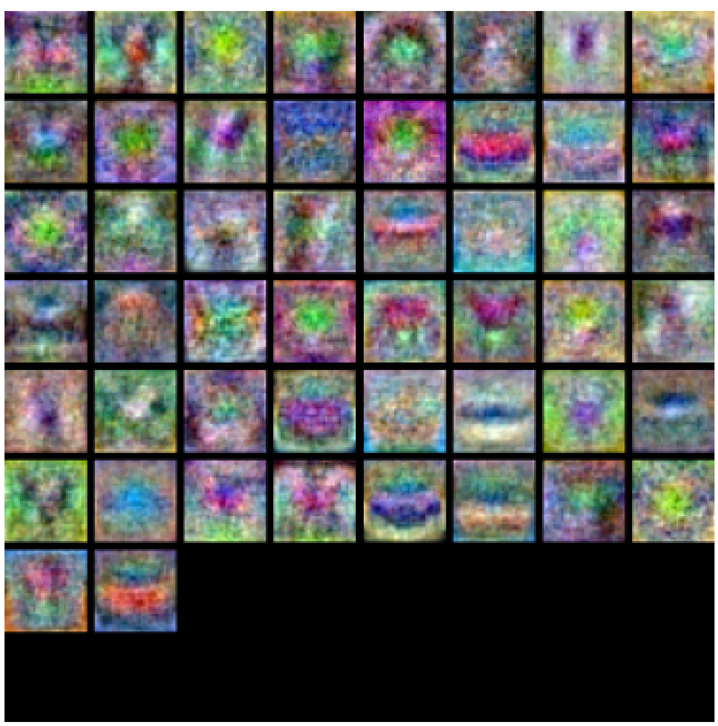
[235]:
```
from utils.vis_utils import visualize_grid

# Visualize the weights of the network

def show_net_weights(net):
    W1 = net.params['W1']
    W1 = W1.T.reshape(32, 32, 3, -1).transpose(3, 0, 1, 2)
    plt.imshow(visualize_grid(W1, padding=3).astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(subopt_net)
show_net_weights(best_net)
```





0.7 Question:

(1) What differences do you see in the weights between the suboptimal net and the best net you arrived at?

0.8 Answer:

(1) The suboptimal weights, when represented as images, are a lot more blurry, whereas the more optimal ones create clearer and more concise “images”.

0.9 Evaluate on test set

[236]:
```
test_acc = (best_net.predict(X_test) == y_test).mean()
print('Test accuracy: ', test_acc)
```

```
Test accuracy:  0.488
```