softmax

December 13, 2024

[1]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'cs231n/cs231n/assignment1'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/cs231n/assignment1/cs231n/datasets
/content/drive/My Drive/cs231n/cs231n/assignment1
```

1 Softmax exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the assignments page on the course website.

This exercise is analogous to the SVM exercise. You will:

- implement a fully-vectorized **loss function** for the Softmax classifier
- implement the fully-vectorized expression for its **analytic gradient**
- **check your implementation** with numerical gradient
- use a validation set to **tune the learning rate and regularization** strength
- **optimize** the loss function with **SGD**
- **visualize** the final learned weights

```
[ ]: import random
     import numpy as np
     from cs231n.data_utils import load_CIFAR10
     import matplotlib.pyplot as plt

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

     # for auto-reloading extenrnal modules
     # see http://stackoverflow.com/questions/1907993/
      ↪autoreload-of-modules-in-ipython
     %load_ext autoreload
     %autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

```
[ ]: def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,␣
      ↪num_dev=500):
         """
         Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
         it for the linear classifier. These are the same steps as we used for the
         SVM, but condensed to a single function.
         """
         # Load the raw CIFAR-10 data
         cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

         # Cleaning up variables to prevent loading data multiple times (which may␣
      ↪cause memory issue)
         try:
            del X_train, y_train
            del X_test, y_test
            print('Clear previously loaded data.')
         except:
            pass

         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]
```

```python
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

    # Preprocessing: reshape the image data into rows
    X_train = np.reshape(X_train, (X_train.shape[0], -1))
    X_val = np.reshape(X_val, (X_val.shape[0], -1))
    X_test = np.reshape(X_test, (X_test.shape[0], -1))
    X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

    # 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
    X_dev -= mean_image

    # add bias dimension and transform into columns
    X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
    X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
    X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
    X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

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


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev = 
 ↪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)
print('Test labels shape: ', y_test.shape)
print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

```
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
```

```
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

1.1 Softmax Classifier

Your code for this section will all be written inside `cs231n/classifiers/softmax.py`.

[ ]:
```python
# First implement the naive softmax loss function with nested loops.
# Open the file cs231n/classifiers/softmax.py and implement the
# softmax_loss_naive function.

from cs231n.classifiers.softmax import softmax_loss_naive
import time

# Generate a random softmax weight matrix and use it to compute the loss.
W = np.random.randn(3073, 10) * 0.0001
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As a rough sanity check, our loss should be something close to -log(0.1).
print('loss: %f' % loss)
print('sanity check: %f' % (-np.log(0.1)))
```

```
loss: 2.358625
sanity check: 2.302585
```

Inline Question 1

Why do we expect our loss to be close to -log(0.1)? Explain briefly.**

$\color{blue}{Your Answer:}$ *(W) score function 0 . iteration loss* $-log(\frac{1}{C})$ *. (C: number of classes) sanity check .*

dw .

$$-log(\frac{e^{s_{y_i}}}{\sum e^{s_j}}) = -s_{y_i} + log(\sum e^{s_j})$$

1) j = y_i

$$\frac{\partial L}{\partial w_j} = -x[j] + \frac{e^{s_j} \cdot x[j]}{\sum e^{s_j}}$$

2) j ≠ y_i

$$\frac{\partial L}{\partial w_j} = \frac{e^{s_j} \cdot x[j]}{\sum e^{s_j}}$$

[ ]:
```python
# Complete the implementation of softmax_loss_naive and implement a (naive)
# version of the gradient that uses nested loops.
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 0.0)

# As we did for the SVM, use numeric gradient checking as a debugging tool.
# The numeric gradient should be close to the analytic gradient.
```

```
from cs231n.gradient_check import grad_check_sparse
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)

# similar to SVM case, do another gradient check with regularization
loss, grad = softmax_loss_naive(W, X_dev, y_dev, 5e1)
f = lambda w: softmax_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
numerical: -0.711033 analytic: -0.711033, relative error: 1.303974e-08
numerical: -3.158884 analytic: -3.158884, relative error: 3.100573e-09
numerical: -0.406902 analytic: -0.406902, relative error: 1.164239e-08
numerical: 0.127734 analytic: 0.127734, relative error: 3.702577e-07
numerical: -0.496821 analytic: -0.496821, relative error: 1.294383e-09
numerical: -2.450904 analytic: -2.450904, relative error: 6.845806e-09
numerical: -0.234934 analytic: -0.234934, relative error: 1.253162e-07
numerical: -1.027032 analytic: -1.027032, relative error: 3.483086e-08
numerical: -1.015511 analytic: -1.015511, relative error: 6.767749e-09
numerical: 1.702807 analytic: 1.702807, relative error: 2.837227e-08
numerical: -5.392010 analytic: -5.392010, relative error: 1.373987e-09
numerical: -1.420173 analytic: -1.420173, relative error: 8.539408e-09
numerical: -0.698757 analytic: -0.698757, relative error: 8.411294e-09
numerical: -1.136044 analytic: -1.136043, relative error: 1.923557e-08
numerical: -1.541982 analytic: -1.541982, relative error: 1.417534e-08
numerical: -3.104776 analytic: -3.104776, relative error: 3.706835e-09
numerical: -1.549034 analytic: -1.549034, relative error: 5.662460e-09
numerical: -2.426142 analytic: -2.426142, relative error: 4.382836e-09
numerical: 0.223683 analytic: 0.223683, relative error: 1.130738e-07
numerical: 4.385243 analytic: 4.385243, relative error: 1.604212e-08
```

[ ]:
```
# Now that we have a naive implementation of the softmax loss function and its␣
↪gradient,
# implement a vectorized version in softmax_loss_vectorized.
# The two versions should compute the same results, but the vectorized version␣
↪should be
# much faster.
tic = time.time()
loss_naive, grad_naive = softmax_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from cs231n.classifiers.softmax import softmax_loss_vectorized
tic = time.time()
loss_vectorized, grad_vectorized = softmax_loss_vectorized(W, X_dev, y_dev, 0.
↪000005)
toc = time.time()
print('vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))
```

```
# As we did for the SVM, we use the Frobenius norm to compare the two versions
# of the gradient.
grad_difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('Loss difference: %f' % np.abs(loss_naive - loss_vectorized))
print('Gradient difference: %f' % grad_difference)
```

```
naive loss: 2.358625e+00 computed in 0.150684s
vectorized loss: 2.358625e+00 computed in 0.011194s
Loss difference: 0.000000
Gradient difference: 0.000000
```

[ ]:
```
# Use the validation set to tune hyperparameters (regularization strength and
# learning rate). You should experiment with different ranges for the learning
# rates and regularization strengths; if you are careful you should be able to
# get a classification accuracy of over 0.35 on the validation set.

from cs231n.classifiers import Softmax
results = {}
best_val = -1
best_softmax = None

################################################################################
# TODO:                                                                        #
# Use the validation set to set the learning rate and regularization strength. #
# This should be identical to the validation that you did for the SVM; save    #
# the best trained softmax classifer in best_softmax.                          #
################################################################################

# Provided as a reference. You may or may not want to change these␣
↪hyperparameters
learning_rates = np.linspace(1e-7, 5e-7, 10)
regularization_strengths = np.linspace(2.5e4, 5e4, 10)

# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

for lr in learning_rates:
  for rs in regularization_strengths:
    softmax = Softmax()
    softmax.train(X_train, y_train, learning_rate=lr, reg=rs, num_iters=1500,␣
↪verbose=False)
    y_train_pred = softmax.predict(X_train)
    train_accuracy = np.mean(y_train == y_train_pred)
    y_val_pred = softmax.predict(X_val)
    val_accuracy = np.mean(y_val == y_val_pred)
    results[(lr, rs)] = (train_accuracy, val_accuracy)
    if best_val < val_accuracy:
```

```python
        best_val = val_accuracy
        best_softmax = softmax

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# Print out results.
for lr, reg in sorted(results):
    train_accuracy, val_accuracy = results[(lr, reg)]
    print('lr %e reg %e train accuracy: %f val accuracy: %f' % (
                lr, reg, train_accuracy, val_accuracy))

print('best validation accuracy achieved during cross-validation: %f' %␣
 ↪best_val)
```

```
lr 1.000000e-07 reg 2.500000e+04 train accuracy: 0.329265 val accuracy: 0.345000
lr 1.000000e-07 reg 2.777778e+04 train accuracy: 0.321837 val accuracy: 0.335000
lr 1.000000e-07 reg 3.055556e+04 train accuracy: 0.322327 val accuracy: 0.342000
lr 1.000000e-07 reg 3.333333e+04 train accuracy: 0.322857 val accuracy: 0.337000
lr 1.000000e-07 reg 3.611111e+04 train accuracy: 0.317327 val accuracy: 0.330000
lr 1.000000e-07 reg 3.888889e+04 train accuracy: 0.314204 val accuracy: 0.322000
lr 1.000000e-07 reg 4.166667e+04 train accuracy: 0.311959 val accuracy: 0.329000
lr 1.000000e-07 reg 4.444444e+04 train accuracy: 0.312469 val accuracy: 0.332000
lr 1.000000e-07 reg 4.722222e+04 train accuracy: 0.306735 val accuracy: 0.318000
lr 1.000000e-07 reg 5.000000e+04 train accuracy: 0.304204 val accuracy: 0.315000
lr 1.444444e-07 reg 2.500000e+04 train accuracy: 0.325878 val accuracy: 0.344000
lr 1.444444e-07 reg 2.777778e+04 train accuracy: 0.324714 val accuracy: 0.338000
lr 1.444444e-07 reg 3.055556e+04 train accuracy: 0.319122 val accuracy: 0.339000
lr 1.444444e-07 reg 3.333333e+04 train accuracy: 0.317776 val accuracy: 0.329000
lr 1.444444e-07 reg 3.611111e+04 train accuracy: 0.307857 val accuracy: 0.332000
lr 1.444444e-07 reg 3.888889e+04 train accuracy: 0.319000 val accuracy: 0.328000
lr 1.444444e-07 reg 4.166667e+04 train accuracy: 0.312837 val accuracy: 0.337000
lr 1.444444e-07 reg 4.444444e+04 train accuracy: 0.313694 val accuracy: 0.322000
lr 1.444444e-07 reg 4.722222e+04 train accuracy: 0.315837 val accuracy: 0.330000
lr 1.444444e-07 reg 5.000000e+04 train accuracy: 0.298000 val accuracy: 0.325000
lr 1.888889e-07 reg 2.500000e+04 train accuracy: 0.331388 val accuracy: 0.346000
lr 1.888889e-07 reg 2.777778e+04 train accuracy: 0.317980 val accuracy: 0.339000
lr 1.888889e-07 reg 3.055556e+04 train accuracy: 0.327408 val accuracy: 0.332000
lr 1.888889e-07 reg 3.333333e+04 train accuracy: 0.311959 val accuracy: 0.331000
lr 1.888889e-07 reg 3.611111e+04 train accuracy: 0.318245 val accuracy: 0.323000
lr 1.888889e-07 reg 3.888889e+04 train accuracy: 0.312286 val accuracy: 0.324000
lr 1.888889e-07 reg 4.166667e+04 train accuracy: 0.311571 val accuracy: 0.329000
lr 1.888889e-07 reg 4.444444e+04 train accuracy: 0.304755 val accuracy: 0.328000
lr 1.888889e-07 reg 4.722222e+04 train accuracy: 0.303531 val accuracy: 0.315000
lr 1.888889e-07 reg 5.000000e+04 train accuracy: 0.306408 val accuracy: 0.319000
lr 2.333333e-07 reg 2.500000e+04 train accuracy: 0.322388 val accuracy: 0.334000
lr 2.333333e-07 reg 2.777778e+04 train accuracy: 0.324980 val accuracy: 0.339000
lr 2.333333e-07 reg 3.055556e+04 train accuracy: 0.319367 val accuracy: 0.344000
```

```
lr 2.333333e-07 reg 3.333333e+04 train accuracy: 0.317592 val accuracy: 0.330000
lr 2.333333e-07 reg 3.611111e+04 train accuracy: 0.316347 val accuracy: 0.328000
lr 2.333333e-07 reg 3.888889e+04 train accuracy: 0.317061 val accuracy: 0.329000
lr 2.333333e-07 reg 4.166667e+04 train accuracy: 0.314449 val accuracy: 0.328000
lr 2.333333e-07 reg 4.444444e+04 train accuracy: 0.308102 val accuracy: 0.324000
lr 2.333333e-07 reg 4.722222e+04 train accuracy: 0.310224 val accuracy: 0.325000
lr 2.333333e-07 reg 5.000000e+04 train accuracy: 0.301061 val accuracy: 0.318000
lr 2.777778e-07 reg 2.500000e+04 train accuracy: 0.332776 val accuracy: 0.343000
lr 2.777778e-07 reg 2.777778e+04 train accuracy: 0.321551 val accuracy: 0.334000
lr 2.777778e-07 reg 3.055556e+04 train accuracy: 0.326429 val accuracy: 0.344000
lr 2.777778e-07 reg 3.333333e+04 train accuracy: 0.315367 val accuracy: 0.335000
lr 2.777778e-07 reg 3.611111e+04 train accuracy: 0.316184 val accuracy: 0.333000
lr 2.777778e-07 reg 3.888889e+04 train accuracy: 0.313571 val accuracy: 0.329000
lr 2.777778e-07 reg 4.166667e+04 train accuracy: 0.304714 val accuracy: 0.320000
lr 2.777778e-07 reg 4.444444e+04 train accuracy: 0.312204 val accuracy: 0.313000
lr 2.777778e-07 reg 4.722222e+04 train accuracy: 0.313061 val accuracy: 0.321000
lr 2.777778e-07 reg 5.000000e+04 train accuracy: 0.294041 val accuracy: 0.315000
lr 3.222222e-07 reg 2.500000e+04 train accuracy: 0.326837 val accuracy: 0.341000
lr 3.222222e-07 reg 2.777778e+04 train accuracy: 0.319837 val accuracy: 0.329000
lr 3.222222e-07 reg 3.055556e+04 train accuracy: 0.318490 val accuracy: 0.340000
lr 3.222222e-07 reg 3.333333e+04 train accuracy: 0.317939 val accuracy: 0.331000
lr 3.222222e-07 reg 3.611111e+04 train accuracy: 0.312633 val accuracy: 0.327000
lr 3.222222e-07 reg 3.888889e+04 train accuracy: 0.324306 val accuracy: 0.332000
lr 3.222222e-07 reg 4.166667e+04 train accuracy: 0.316796 val accuracy: 0.331000
lr 3.222222e-07 reg 4.444444e+04 train accuracy: 0.309163 val accuracy: 0.322000
lr 3.222222e-07 reg 4.722222e+04 train accuracy: 0.307653 val accuracy: 0.323000
lr 3.222222e-07 reg 5.000000e+04 train accuracy: 0.287673 val accuracy: 0.303000
lr 3.666667e-07 reg 2.500000e+04 train accuracy: 0.322837 val accuracy: 0.337000
lr 3.666667e-07 reg 2.777778e+04 train accuracy: 0.322122 val accuracy: 0.333000
lr 3.666667e-07 reg 3.055556e+04 train accuracy: 0.328796 val accuracy: 0.344000
lr 3.666667e-07 reg 3.333333e+04 train accuracy: 0.323959 val accuracy: 0.338000
lr 3.666667e-07 reg 3.611111e+04 train accuracy: 0.315408 val accuracy: 0.331000
lr 3.666667e-07 reg 3.888889e+04 train accuracy: 0.302816 val accuracy: 0.317000
lr 3.666667e-07 reg 4.166667e+04 train accuracy: 0.314061 val accuracy: 0.331000
lr 3.666667e-07 reg 4.444444e+04 train accuracy: 0.316347 val accuracy: 0.335000
lr 3.666667e-07 reg 4.722222e+04 train accuracy: 0.306469 val accuracy: 0.323000
lr 3.666667e-07 reg 5.000000e+04 train accuracy: 0.313673 val accuracy: 0.328000
lr 4.111111e-07 reg 2.500000e+04 train accuracy: 0.321245 val accuracy: 0.330000
lr 4.111111e-07 reg 2.777778e+04 train accuracy: 0.323163 val accuracy: 0.337000
lr 4.111111e-07 reg 3.055556e+04 train accuracy: 0.326776 val accuracy: 0.338000
lr 4.111111e-07 reg 3.333333e+04 train accuracy: 0.308286 val accuracy: 0.322000
lr 4.111111e-07 reg 3.611111e+04 train accuracy: 0.312857 val accuracy: 0.333000
lr 4.111111e-07 reg 3.888889e+04 train accuracy: 0.303918 val accuracy: 0.322000
lr 4.111111e-07 reg 4.166667e+04 train accuracy: 0.312367 val accuracy: 0.328000
lr 4.111111e-07 reg 4.444444e+04 train accuracy: 0.307408 val accuracy: 0.318000
lr 4.111111e-07 reg 4.722222e+04 train accuracy: 0.310000 val accuracy: 0.326000
lr 4.111111e-07 reg 5.000000e+04 train accuracy: 0.305469 val accuracy: 0.319000
lr 4.555556e-07 reg 2.500000e+04 train accuracy: 0.325939 val accuracy: 0.346000
```

```
lr 4.555556e-07 reg 2.777778e+04 train accuracy: 0.321633 val accuracy: 0.332000
lr 4.555556e-07 reg 3.055556e+04 train accuracy: 0.321755 val accuracy: 0.329000
lr 4.555556e-07 reg 3.333333e+04 train accuracy: 0.321143 val accuracy: 0.332000
lr 4.555556e-07 reg 3.611111e+04 train accuracy: 0.319041 val accuracy: 0.328000
lr 4.555556e-07 reg 3.888889e+04 train accuracy: 0.317796 val accuracy: 0.328000
lr 4.555556e-07 reg 4.166667e+04 train accuracy: 0.310306 val accuracy: 0.321000
lr 4.555556e-07 reg 4.444444e+04 train accuracy: 0.289163 val accuracy: 0.311000
lr 4.555556e-07 reg 4.722222e+04 train accuracy: 0.310429 val accuracy: 0.325000
lr 4.555556e-07 reg 5.000000e+04 train accuracy: 0.294061 val accuracy: 0.312000
lr 5.000000e-07 reg 2.500000e+04 train accuracy: 0.328796 val accuracy: 0.345000
lr 5.000000e-07 reg 2.777778e+04 train accuracy: 0.330245 val accuracy: 0.350000
lr 5.000000e-07 reg 3.055556e+04 train accuracy: 0.323347 val accuracy: 0.336000
lr 5.000000e-07 reg 3.333333e+04 train accuracy: 0.324816 val accuracy: 0.340000
lr 5.000000e-07 reg 3.611111e+04 train accuracy: 0.315184 val accuracy: 0.339000
lr 5.000000e-07 reg 3.888889e+04 train accuracy: 0.316082 val accuracy: 0.324000
lr 5.000000e-07 reg 4.166667e+04 train accuracy: 0.303776 val accuracy: 0.317000
lr 5.000000e-07 reg 4.444444e+04 train accuracy: 0.304327 val accuracy: 0.319000
lr 5.000000e-07 reg 4.722222e+04 train accuracy: 0.292286 val accuracy: 0.326000
lr 5.000000e-07 reg 5.000000e+04 train accuracy: 0.303367 val accuracy: 0.314000
best validation accuracy achieved during cross-validation: 0.350000
```

[ ]:
```python
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.341000
```

Inline Question 2 - *True or False*

Suppose the overall training loss is defined as the sum of the per-datapoint loss over all training examples. It is possible to add a new datapoint to a training set that would leave the SVM loss unchanged, but this is not the case with the Softmax classifier loss.

$YourAnswer:$

$YourExplanation:$ Softmax classifier loss .

$Loss_{softmax} : -log(\frac{e^{s_{y_i}}}{\sum e^{s_j}})$

SVM loss s_{y_i} ( ) s_j( ) margin loss .($s_j > s_{y_i}$) , $s_{y_i} > s_j$, margin loss 0 . , loss .

$Loss_{svm} : \frac{1}{n} \sum max(0, s_j - s_{y_i} + margin)$

[ ]:
```python
# Visualize the learned weights for each class
w = best_softmax.W[:-1,:] # strip out the bias
w = w.reshape(32, 32, 3, 10)

w_min, w_max = np.min(w), np.max(w)
```

```python
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
for i in range(10):
    plt.subplot(2, 5, i + 1)

    # Rescale the weights to be between 0 and 255
    wimg = 255.0 * (w[:, :, :, i].squeeze() - w_min) / (w_max - w_min)
    plt.imshow(wimg.astype('uint8'))
    plt.axis('off')
    plt.title(classes[i])
```