cattern

February 28, 2026

1 neural_net.py

[1]:
```python
from __future__ import print_function

import numpy as np
```

[ ]:
```python
class TwoLayerNet(object):
  """
  A two-layer fully-connected neural network. The net has an input dimension of
  N, a hidden layer dimension of H, and performs classification over C classes.
  We train the network with a softmax loss function and L2 regularization on the
  weight matrices. The network uses a ReLU nonlinearity after the first fully
  connected layer.

  In other words, the network has the following architecture:

  input - fully connected layer - ReLU - fully connected layer - softmax

  The outputs of the second fully-connected layer are the scores for each class.
  """

  def __init__(self, input_size, hidden_size, output_size, std=1e-4):
    """
    Initialize the model. Weights are initialized to small random values and
    biases are initialized to zero. Weights and biases are stored in the
    variable self.params, which is a dictionary with the following keys:

    W1: First layer weights; has shape (D, H)
    b1: First layer biases; has shape (H,)
    W2: Second layer weights; has shape (H, C)
    b2: Second layer biases; has shape (C,)

    Inputs:
    - input_size: The dimension D of the input data.
    - hidden_size: The number of neurons H in the hidden layer.
    - output_size: The number of classes C.
    """
```

```python
        self.params = {}
        self.params["W1"] = std * np.random.randn(input_size, hidden_size)
        self.params["b1"] = np.zeros(hidden_size)
        self.params["W2"] = std * np.random.randn(hidden_size, output_size)
        self.params["b2"] = np.zeros(output_size)


    def loss(self, X, y=None, reg=0.0):
        """
        Compute the loss and gradients for a two layer fully connected neural␣
 ↪network.

        Inputs:
        - X: Input data of shape (N, D). Each X[i] is a training sample.
        - y: Vector of training labels. y[i] is the label for X[i], and each y[i] is
          an integer in the range 0 <= y[i] < C. This parameter is optional; if it
          is not passed then we only return scores, and if it is passed then we
          instead return the loss and gradients.
        - reg: Regularization strength.

        Returns:
        If y is None, return a matrix scores of shape (N, C) where scores[i, c] is
        the score for class c on input X[i].

        If y is not None, instead return a tuple of:
        - loss: Loss (data loss and regularization loss) for this batch of training
          samples.
        - grads: Dictionary mapping parameter names to gradients of those parameters
          with respect to the loss function; has the same keys as self.params.
        """
        # Unpack variables from the params dictionary
        W1, b1 = self.params["W1"], self.params["b1"]
        W2, b2 = self.params["W2"], self.params["b2"]
        N, _ = X.shape

        #############################################################################
        # T5: Perform the forward pass, computing the class scores for the input.   #
        # Store the result in the scores variable, which should be an array of      #
        # shape (N, C). Note that this does not include the softmax                 #
        # HINT: This is just a series of matrix multiplication.                     #
        #############################################################################

        # Compute the forward pass
        X1 = X @ W1 + b1
        X1 = np.maximum(0, X1)
        scores = X1 @ W2 + b2
```

```python
    ################################################################################
    #                                  END OF T5                                   #
    ################################################################################

    # If the targets are not given then jump out, we're done
    if y is None:
      return scores

    ################################################################################
    # T6: Finish the forward pass, and compute the loss. This should include      #
    # both the data loss and L2 regularization for W1 and W2. Store the result #
    # in the variable loss, which should be a scalar. Use the Softmax              #
    # classifier loss.                                                             #
    ################################################################################

    # Compute the loss
    scores_shifted  = scores - np.max(scores, axis=1, keepdims=True)
    scores_softmax  = np.exp(scores_shifted)
    scores_softmax /= np.sum(scores_softmax, axis=1, keepdims=True)

    loss  = -np.sum(np.log(scores_softmax[np.arange(N), y]))
    loss /= N
    loss += 1/2 * reg * ((W1 * W1).sum() + (W2 * W2).sum())

    ################################################################################
    #                                  END OF T6                                   #
    ################################################################################


    ################################################################################
    # T7: Compute the backward pass, computing derivatives of the weights         #
    # and biases. Store the results in the grads dictionary. For example,         #
    # grads["W1"] should store the gradient on W1, and be a matrix of same         #
    # size don't forget about the regularization term                              #
    ################################################################################

    # Backward Pass: Compute Gradients
    grads = {}

    dscores = scores_softmax
    dscores[np.arange(N), y] -= 1
    dscores /= N

    grads["W2"] = X1.T @ dscores + reg * W2
    grads["b2"] = np.sum(dscores, axis=0)

    dX1 = dscores @ W2.T
```

```python
    dX1[X1 <= 0] = 0

    grads["W1"] = X.T @ dX1 + reg * W1
    grads["b1"] = np.sum(dX1, axis=0)

    ############################################################################
    #                              END OF T7                                   #
    ############################################################################

    return loss, grads


  def train(
    self,
    X, y, X_val, y_val,
    learning_rate=1e-3,
    learning_rate_decay=0.95,
    reg=5e-6,
    num_iters=100,
    batch_size=200,
    verbose=False
  ):
    """
    Train this neural network using stochastic gradient descent.

    Inputs:
    - X: A numpy array of shape (N, D) giving training data.
    - y: A numpy array of shape (N,) giving training labels; y[i] = c means that
      X[i] has label c, where 0 <= c < C.
    - X_val: A numpy array of shape (N_val, D) giving validation data.
    - y_val: A numpy array of shape (N_val,) giving validation labels.
    - learning_rate: Scalar giving learning rate for optimization.
    - learning_rate_decay: Scalar giving factor used to decay the learning rate
      after each epoch.
    - reg: Scalar giving regularization strength.
    - num_iters: Number of steps to take when optimizing.
    - batch_size: Number of training examples to use per step.
    - verbose: boolean; if true print progress during optimization.
    """

    num_train = X.shape[0]
    iterations_per_epoch = max(num_train / batch_size, 1)

    # Use SGD to optimize the parameters in self.model
    loss_history = []
    train_acc_history = []
    val_acc_history = []
```

```python
for it in range(num_iters):
  #########################################################################
  # T8: Create a random minibatch of training data and labels, storing    #
  # them in X_batch and y_batch respectively.                             #
  # You might find np.random.choice() helpful.                            #
  #########################################################################

  random_mask = np.random.choice(num_train, batch_size, replace=True)
  X_batch = X[random_mask]
  y_batch = y[random_mask]

  #########################################################################
  #                             END OF YOUR T8                            #
  #########################################################################

  # Compute loss and gradients using the current minibatch
  loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
  loss_history.append(loss)

  #########################################################################
  # T9: Use the gradients in the grads dictionary to update the           #
  # parameters of the network (stored in the dictionary self.params)      #
  # using stochastic gradient descent. You'll need to use the gradients   #
  # stored in the grads dictionary defined above.                         #
  #########################################################################

  self.params["W1"] -= learning_rate * grads["W1"]
  self.params["b1"] -= learning_rate * grads["b1"]

  self.params["W2"] -= learning_rate * grads["W2"]
  self.params["b2"] -= learning_rate * grads["b2"]

  #########################################################################
  #                             END OF YOUR T9                            #
  #########################################################################

  if verbose and it % 100 == 0:
    print("iteration %d / %d: loss %f" % (it, num_iters, loss))

  # Every epoch, check train and val accuracy and decay learning rate.
  if it % iterations_per_epoch == 0:
    # Check accuracy
    train_acc = (self.predict(X_batch) == y_batch).mean()
    val_acc = (self.predict(X_val) == y_val).mean()
    train_acc_history.append(train_acc)
    val_acc_history.append(val_acc)
```

```python
        #########################################################################
        # T10: Decay learning rate (exponentially) after each epoch            #
        #########################################################################

        # Decay learning rate
        learning_rate *= learning_rate_decay

        #########################################################################
        #                              END OF YOUR T10                          #
        #########################################################################


    return {
      "loss_history": loss_history,
      "train_acc_history": train_acc_history,
      "val_acc_history": val_acc_history,
    }


  def predict(self, X):
    """
    Use the trained weights of this two-layer network to predict labels for
    data points. For each data point we predict scores for each of the C
    classes, and assign each data point to the class with the highest score.

    Inputs:
    - X: A numpy array of shape (N, D) giving N D-dimensional data points to
      classify.

    Returns:
    - y_pred: A numpy array of shape (N,) giving predicted labels for each of
      the elements of X. For all i, y_pred[i] = c means that X[i] is predicted
      to have class c, where 0 <= c < C.
    """

    ###########################################################################
    # T11: Implement this function; it should be VERY simple!                 #
    ###########################################################################

    scores = np.array(self.loss(X))
    y_pred = np.argmax(scores, axis=1)

    ###########################################################################
    #                              END OF YOUR T11                            #
    ###########################################################################
```

```
    return y_pred
```

---

2 data_utils.py

[3]:
```
import os
import pandas as pd

from glob import glob
from functools import reduce
from sklearn.model_selection import train_test_split

article_types = ["article", "encyclopedia", "news", "novel"]
```

[ ]:
```
def generate_words(files):
    """
    Transform list of files to list of words,
    removing new line character
    and replace name entity "<NE>...</NE>" and abbreviation "<AB>...</AB>"␣
 ↪symbol
    """

    repls = {"<NE>" : "","</NE>" : "","<AB>": "","</AB>": ""}

    words_all = []
    for _, file in enumerate(files):
        lines = open(file, "r")
        for line in lines:
            line = reduce(lambda a, kv: a.replace(*kv), repls.items(), line)
            words = [word for word in line.split("|") if word is not "\n"]
            words_all.extend(words)

    return words_all
```

[5]:
```
def create_char_dataframe(words):
    """
    Give list of input tokenized words,
    create dataframe of characters where first character of
    the word is tagged as 1, otherwise 0
    Example
    =======
    [" ", "  "] to dataframe of
    [{"char": " ", "type": ..., "target": 1}, ...,
     {"char": " ", "type": ..., "target": 0}]
    """
```

```
    char_dict = []
    for word in words:
        for i, char in enumerate(word):
            if i == 0:
                char_dict.append({"char": char, "target": True})
            else:
                char_dict.append({"char": char, "target": False})

    return pd.DataFrame(char_dict)
```

[6]:
```
def generate_best_dataset(best_path, output_path="cleaned_data",␣
↪create_val=False):
    """
    Generate CSV file for training and testing data
    Input
    =====
    best_path: str, path to BEST folder which contains unzipped subfolder
        "article", "encyclopedia", "news", "novel"
    cleaned_data: str, path to output folder, the cleaned data will be saved
        in the given folder name where training set will be stored in `train`␣
↪folder
        and testing set will be stored on `test` folder
    create_val: boolean, True or False, if True, divide training set into␣
↪training set and
        validation set in `val` folder
    """

    if not os.path.isdir(output_path):
        os.mkdir(output_path)

    if not os.path.isdir(os.path.join(output_path, "train")):
        os.makedirs(os.path.join(output_path, "train"))

    if not os.path.isdir(os.path.join(output_path, "test")):
        os.makedirs(os.path.join(output_path, "test"))

    if not os.path.isdir(os.path.join(output_path, "val")) and create_val:
        os.makedirs(os.path.join(output_path, "val"))

    for article_type in article_types:
        files = glob(os.path.join(best_path, article_type, "*.txt"))
        files_train, files_test = train_test_split(files, random_state=0,␣
↪test_size=0.1)

        if create_val:
```

```
            files_train, files_val = train_test_split(files_train,␣
↪random_state=0, test_size=0.1)
            val_words = generate_words(files_val)
            val_df = create_char_dataframe(val_words)
            val_df.to_csv(os.path.join(output_path, "val", "df_best_{}_val.csv".
↪format(article_type)), index=False)

        train_words = generate_words(files_train)
        test_words = generate_words(files_test)
        train_df = create_char_dataframe(train_words)
        test_df = create_char_dataframe(test_words)

        train_df.to_csv(os.path.join(output_path, "train", "df_best_{}_train.
↪csv".format(article_type)), index=False)
        test_df.to_csv(os.path.join(output_path, "test", "df_best_{}_test.csv".
↪format(article_type)), index=False)

        print("Save {} to CSV file".format(article_type))
```

---

3 gradient_check.py

[7]:
```
from __future__ import print_function

import numpy as np
from random import randrange
```

[8]:
```
def eval_numerical_gradient(f, x, verbose=True, h=0.00001):
  """
  A naive implementation of numerical gradient of f at x.
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient at
  """

  # Evaluate function value at original point.
  fx = f(x)
  grad = np.zeros_like(x)

  # Iterate over all indexes in x.
  it = np.nditer(x, flags=["multi_index"], op_flags=[["readwrite"]])

  while not it.finished:
    # Evaluate function at x + h.
    ix = it.multi_index
    oldval = x[ix]
```

```
    # Increment by h.
    x[ix] = oldval + h

    # Evalute f(x + h).
    fxph = f(x)
    x[ix] = oldval - h

    # Evalute f(x - h).
    fxmh = f(x)

    # Restore the original value.
    x[ix] = oldval

    # Compute the partial derivative with centered formula.
    grad[ix] = (fxph - fxmh) / (2 * h) # The slope

    if verbose:
      print(ix, grad[ix])

    # Step to next dimension
    it.iternext()

  return grad
```

[9]:
```
def eval_numerical_gradient_array(f, x, df, h=1e-5):
  """
  Evaluate a numeric gradient for a function that accepts a numpy
  array and returns a numpy array.
  """

  grad = np.zeros_like(x)
  it = np.nditer(x, flags=["multi_index"], op_flags=[["readwrite"]])

  while not it.finished:
    ix = it.multi_index

    oldval = x[ix]
    x[ix] = oldval + h

    pos = f(x).copy()
    x[ix] = oldval - h

    neg = f(x).copy()
    x[ix] = oldval

    grad[ix] = np.sum((pos - neg) * df) / (2 * h)
```

```
    it.iternext()

  return grad
```

[10]:
```
def eval_numerical_gradient_blobs(f, inputs, output, h=1e-5):
  """
  Compute numeric gradients for a function that operates on input
  and output blobs.

  We assume that f accepts several input blobs as arguments, followed by a blob
  into which outputs will be written. For example, f might be called like this:

  f(x, w, out)

  where x and w are input Blobs, and the result of f will be written to out.

  Inputs:
  - f: function
  - inputs: tuple of input blobs
  - output: output blob
  - h: step size
  """
  numeric_diffs = []
  for input_blob in inputs:
    diff = np.zeros_like(input_blob.diffs)

    it = np.nditer(input_blob.vals, flags=["multi_index"],␣
↪op_flags=[["readwrite"]])

    while not it.finished:
      idx = it.multi_index
      orig = input_blob.vals[idx]

      input_blob.vals[idx] = orig + h

      f(*(inputs + (output,)))

      pos = np.copy(output.vals)
      input_blob.vals[idx] = orig - h

      f(*(inputs + (output,)))

      neg = np.copy(output.vals)
      input_blob.vals[idx] = orig

      diff[idx] = np.sum((pos - neg) * output.diffs) / (2.0 * h)
```

```
      it.iternext()

    numeric_diffs.append(diff)

  return numeric_diffs
```

[11]:
```
def eval_numerical_gradient_net(net, inputs, output, h=1e-5):
  return eval_numerical_gradient_blobs(lambda *args: net.forward(), inputs,␣
 ↪output, h=h)
```

[12]:
```
def grad_check_sparse(f, x, analytic_grad, num_checks=10, h=1e-5):
  """
  Sample a few random elements and only return numerical
  gradients in these dimensions.
  """

  for _ in range(num_checks):
    ix = tuple([randrange(m) for m in x.shape])

    oldval = x[ix]

    # Increment by h.
    x[ix] = oldval + h

    # Evaluate f(x + h).
    fxph = f(x)

    # Decrement by h.
    x[ix] = oldval - h

    # Evaluate f(x - h).
    fxmh = f(x)

    # Restore the original value.
    x[ix] = oldval

    grad_numerical = (fxph - fxmh) / (2 * h)
    grad_analytic = analytic_grad[ix]
    rel_error = abs(grad_numerical - grad_analytic) / (abs(grad_numerical) +␣
 ↪abs(grad_analytic))

    print("numerical: %f analytic: %f, relative error: %e" % (grad_numerical,␣
 ↪grad_analytic, rel_error))
```
