CS 6384 Computer Vision Homework 4 *

Professor Yapeng Tian

Download the homework4_programming.zip file from eLearning. Finish the following programming problems and submit your scripts to eLearning. You can zip all files for submission. Our TA will run your scripts to verify them. **Do not include the CIFAR10 dataset in the submission.**

Install the Python packages needed by

- pip install -r requirement.txt

Here are some useful resources:

- Python basics `https://pythonbasics.org/`
- Numpy `https://numpy.org/doc/stable/user/basics.html`
- OpenCV `https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html`

For this homework, you cannot use any deep learning libraries such as PyTorch or TensorFlow.

When using back-propagation to train neural networks, we compute local gradients of each layer and combine them to learn the weights in the neural networks. Fig. 1 illustrates this process. In this example, we have a layer that takes a matrix (tensor) x with dimension $D_x \times M_x$ and a matrix (tensor) y with dimension $D_y \times M_y$ as input. The output of this layer is z with dimension $D_z \times M_z$.

In the forward function, this layer computes the output z given x and y. It can also output a *cache* object that contains all the values needed during back-propagation.

In the backward function, this layer receives the *upstream gradients* and the *cache* object, and compute the *downstream gradients*. In this example, the upstream gradients is $\frac{\partial L}{\partial z}$, where L denotes the final loss function of the network. Note that the loss function outputs a scalar. Therefore, $\frac{\partial L}{\partial z}$ is with dimension $D_z \times M_z$ that is the same as z. We use chain rule to compute the downstream gradients

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}, \frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y},$$

*This homework is adapted from Dr. Justin Johnson at the University of Michigan

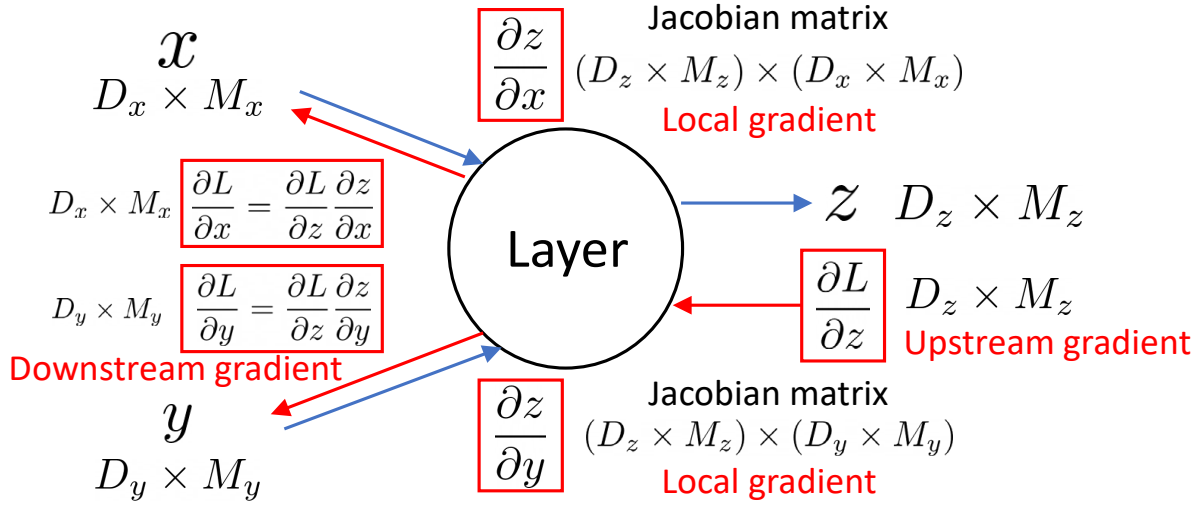

Figure 1: Back-propagation of gradients.

where $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$ are the local gradients in this layer. They are the Jacobian matrices:

$$(\frac{\partial z}{\partial x})_{ij} = \frac{\partial z_i}{\partial x_j}, (\frac{\partial z}{\partial y})_{ij} = \frac{\partial z_i}{\partial y_j}.$$

We can consider $\frac{\partial z}{\partial x}$ as a matrix with dimension $(D_z \times M_z) \times (D_x \times M_x)$. Then we can do a matrix-vector multiplication to compute $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$, which have the same dimension as x. Similarly, we can compute $\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$.

Problem 1

(2 points) Back-propagation.

In this question, you need to implement the fully-connected layer, the ReLU layer, the softmax loss function and the L2 regularization loss function in neuralnet/layers.py.

After your implementation, you need to use the script neuralnet/gradcheck layers.py to perform numeric gradient checking on your implementations. Given a function $f : \mathcal{R} \to \mathcal{R}$, we can approximate the gradient of f at a point $x_0 \in \mathcal{R}$ using central difference:

$$\frac{\partial f}{\partial x}(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h}. \tag{1.1}$$

The difference between all numeric and analytic gradients should be less than 10^{-9}. Keep in mind that numeric gradient checking does not check whether you have correctly implemented the forward pass. It only checks whether the backward pass you have implemented actually computes the gradient of the forward pass that you implemented.

(1.1) Fully connected layer. Implement the fc_forward() function and the fc_backward() function in neuralnet/layers.py after reading the following derivation.

The input to a FC layer is a tensor x with shape (N, D_x), where N is the batch size and D_x is the dimension of the feature. The weight matrix W in the FC layer is with shape (D_x, D_w) and the bias b of the FC layer is a vector with dimension D_w. The output of the FC layer is a tensor y with shape (N, D_w). The $i$the row of the y matrix is computed by

$$y_i = x_i W + b, \tag{1.2}$$

where y_i and x_i are the ith row of y and x, respectively, i.e., the ith data point in the batch. With all the data points, we have

$$\begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,D_w} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,D_w} \\ \vdots & \vdots & \vdots & \vdots \\ y_{N,1} & y_{N,2} & \cdots & y_{N,D_w} \end{bmatrix} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,D_x} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,D_x} \\ \vdots & \vdots & \vdots & \vdots \\ x_{N,1} & x_{N,2} & \cdots & x_{N,D_x} \end{bmatrix} \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,D_w} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,D_w} \\ \vdots & \vdots & \vdots & \vdots \\ w_{D_x,1} & w_{D_x,2} & \cdots & w_{D_x,D_w} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 & \cdots & b_{D_w} \\ b_1 & b_2 & \cdots & b_{D_w} \\ \vdots & \vdots & \vdots & \vdots \\ b_1 & b_2 & \cdots & b_{D_w} \end{bmatrix}. \tag{1.3}$$

In the backward function of the FC layer, we receive upstream gradients $\frac{\partial L}{\partial y}$ with shape (N, D_w). We need to compute the downstream gradients $\frac{\partial L}{\partial x}$, $\frac{\partial L}{\partial W}$ and $\frac{\partial L}{\partial b}$. To do so, let's first consider

$$\frac{\partial L}{\partial x} = \sum_{i=1}^{N} \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{i,j}} \cdot \frac{\partial y_{i,j}}{\partial x} \tag{1.4}$$

$$= \sum_{i=1}^{N} \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{i,j}} \begin{bmatrix} 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ w_{1,j} & w_{2,j} & \cdots & w_{D_x,j} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \text{(the ith row)} \tag{1.5}$$

$$= \sum_{j=1}^{D_w} \begin{bmatrix} \frac{\partial L}{\partial y_{1,j}} w_{1,j} & \frac{\partial L}{\partial y_{1,j}} w_{2,j} & \cdots & \frac{\partial L}{\partial y_{1,j}} w_{D_x,j} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial L}{\partial y_{i,j}} w_{1,j} & \frac{\partial L}{\partial y_{i,j}} w_{2,j} & \cdots & \frac{\partial L}{\partial y_{i,j}} w_{D_x,j} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial L}{\partial y_{N,j}} w_{1,j} & \frac{\partial L}{\partial y_{N,j}} w_{2,j} & \cdots & \frac{\partial L}{\partial y_{N,j}} w_{D_x,j} \end{bmatrix} \tag{1.6}$$

$$= \begin{bmatrix} \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{1,j}} w_{1,j} & \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{1,j}} w_{2,j} & \cdots & \frac{\partial L}{\partial y_{1,j}} w_{D_x,j} \\ \vdots & \vdots & \vdots & \vdots \\ \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{i,j}} w_{1,j} & \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{i,j}} w_{2,j} & \cdots & \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{i,j}} w_{D_x,j} \\ \vdots & \vdots & \vdots & \vdots \\ \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{N,j}} w_{1,j} & \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{N,j}} w_{2,j} & \cdots & \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{N,j}} w_{D_x,j} \end{bmatrix} \tag{1.7}$$

$$= \frac{\partial L}{\partial y} W^T. \tag{1.8}$$

We have a compact formula to compute the downstream gradients of x as

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} W^T. \tag{1.9}$$

Similarly, we have

$$\frac{\partial L}{\partial W} = \sum_{i=1}^{N}\sum_{j=1}^{D_w}\frac{\partial L}{\partial y_{i,j}}\cdot\frac{\partial y_{i,j}}{\partial W} \tag{1.10}$$

$$= \sum_{i=1}^{N}\sum_{j=1}^{D_w}\frac{\partial L}{\partial y_{i,j}}\underbrace{\begin{bmatrix}0 & \cdots & x_{i,1} & \cdots & 0\\ 0 & \cdots & x_{i,2} & \cdots & 0\\ \vdots & \vdots & \vdots & \vdots & \vdots\\ 0 & \cdots & x_{i,D_x} & \cdots & 0\end{bmatrix}}_{\text{the jth column}} \tag{1.11}$$

$$= \sum_{i=1}^{N}\begin{bmatrix}\frac{\partial L}{\partial y_{i,1}}x_{i,1} & \cdots & \frac{\partial L}{\partial y_{i,j}}x_{i,1} & \cdots & \frac{\partial L}{\partial y_{i,D_w}}x_{i,1}\\ \frac{\partial L}{\partial y_{i,1}}x_{i,2} & \cdots & \frac{\partial L}{\partial y_{i,j}}x_{i,2} & \cdots & \frac{\partial L}{\partial y_{i,D_w}}x_{i,2}\\ \vdots & \vdots & \vdots & \vdots & \vdots\\ \frac{\partial L}{\partial y_{i,1}}x_{i,D_x} & \cdots & \frac{\partial L}{\partial y_{i,j}}x_{i,D_x} & \cdots & \frac{\partial L}{\partial y_{i,D_w}}x_{i,D_x}\end{bmatrix} \tag{1.12}$$

$$= \begin{bmatrix}\sum_{i=1}^{N}\frac{\partial L}{\partial y_{i,1}}x_{i,1} & \cdots & \sum_{i=1}^{N}\frac{\partial L}{\partial y_{i,j}}x_{i,1} & \cdots & \sum_{i=1}^{N}\frac{\partial L}{\partial y_{i,D_w}}x_{i,1}\\ \sum_{i=1}^{N}\frac{\partial L}{\partial y_{i,1}}x_{i,2} & \cdots & \sum_{i=1}^{N}\frac{\partial L}{\partial y_{i,j}}x_{i,2} & \cdots & \sum_{i=1}^{N}\frac{\partial L}{\partial y_{i,D_w}}x_{i,2}\\ \vdots & \vdots & \vdots & \vdots & \vdots\\ \sum_{i=1}^{N}\frac{\partial L}{\partial y_{i,1}}x_{i,D_x} & \cdots & \sum_{i=1}^{N}\frac{\partial L}{\partial y_{i,j}}x_{i,D_x} & \cdots & \sum_{i=1}^{N}\frac{\partial L}{\partial y_{i,D_w}}x_{i,D_x}\end{bmatrix} \tag{1.13}$$

$$= x^T\frac{\partial L}{\partial y}. \tag{1.14}$$

Therefore,

$$\frac{\partial L}{\partial W} = x^T\frac{\partial L}{\partial y}. \tag{1.15}$$

Lastly, we compute

$$\frac{\partial L}{\partial b} = \sum_{i=1}^{N}\sum_{j=1}^{D_w}\frac{\partial L}{\partial y_{i,j}}\cdot\frac{\partial y_{i,j}}{\partial b} \tag{1.16}$$

$$= \sum_{i=1}^{N}\sum_{j=1}^{D_w}\frac{\partial L}{\partial y_{i,j}}\underbrace{\begin{bmatrix}0 & \cdots & 1 & \cdots & 0\end{bmatrix}}_{\text{the jth column}} \tag{1.17}$$

$$= \sum_{i=1}^{N}\begin{bmatrix}\frac{\partial L}{\partial y_{i,1}} & \cdots & \frac{\partial L}{\partial y_{i,j}} & \cdots & \frac{\partial L}{\partial y_{i,D_w}}\end{bmatrix} \tag{1.18}$$

$$= \begin{bmatrix}\sum_{i=1}^{N}\frac{\partial L}{\partial y_{i,1}} & \cdots & \sum_{i=1}^{N}\frac{\partial L}{\partial y_{i,j}} & \cdots & \sum_{i=1}^{N}\frac{\partial L}{\partial y_{i,D_w}}\end{bmatrix} \tag{1.19}$$

$$= \mathbf{1}^T\frac{\partial L}{\partial y}. \tag{1.20}$$

That is

$$\frac{\partial L}{\partial b} = \mathbf{1}^T\frac{\partial L}{\partial y}, \tag{1.21}$$

where $\mathbf{1}$ denotes a column vector with all 1s.

(1.2) ReLU layer. Implement the relu_forward() function and the relu_backward() function in neuralnet/layers.py. The ReLU activation function is defined as

$$\text{ReLU}(x) = \max(0, x) \tag{1.22}$$

$$= \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise,} \end{cases} \tag{1.23}$$

for each element in a tensor.

(1.3) Softmax Loss Function. Implement the softmax_loss() function in neuralnet/layers.py after reading the following material.

The input to a softmax loss function layer is a tensor x with shape (N, C), where N is the batch size and C is the number of categories to be classified. The softmax loss function first converts the scores x into a set of N probability distributions over the categories, defined as:

$$p_{i,c} = \frac{\exp(x_{i,c})}{\sum_{j=1}^{C} \exp(x_{i,j})}, i = 1, 2, \ldots, N, c = 1, 2, \ldots, C. \tag{1.24}$$

Then the softmax loss function is defined as

$$L = -\frac{1}{N} \sum_{i=1}^{N} \log(p_{i,y_i}), \tag{1.25}$$

where $y_i \in \{1, 2, \ldots, C\}$ is the ground truth label for the ith data point.

A naive implementation of the softmax loss function can result in numeric instability when the value of some $x_{i,c}$ in Eq. (1.24) is large. Then it can cause overflow with $\exp(x_{i,c})$. To avoid this, we can compute the probabilities by

$$p_{i,c} = \frac{\exp(z_{i,c})}{\sum_{j=1}^{C} \exp(z_{i,j})} = \frac{\exp(x_{i,c} - M_i)}{\sum_{j=1}^{C} \exp(x_{i,j} - M_i)}, i = 1, 2, \ldots, N, c = 1, 2, \ldots, C, \tag{1.26}$$

where $M_i = \max_c x_{i,c}$, i.e., the maximum score for data point i among the categories, and $z_{i,c} = x_{i,c} - M_i$. By doing so, we can avoid overflow with the exponential. It is not hard to see that

$$p_{i,c} = \frac{\exp(x_{i,c} - M_i)}{\sum_{j=1}^{C} \exp(x_{i,j} - M_i)} = \frac{\exp(x_{i,c}) \exp(-M_i)}{\sum_{j=1}^{C} \exp(x_{i,j}) \exp(-M_i)} = \frac{\exp(x_{i,c})}{\sum_{j=1}^{C} \exp(x_{i,j})}. \tag{1.27}$$

Your softmax implementation should use this max-subtraction trick for numeric stability. You can run the script neuralnet/check softmax stability.py to check the numeric stability of your softmax loss implementation.

In the backward function of the softmax loss function, we need to compute the downstream gradients $\frac{\partial L}{\partial x}$ with shape (N, C). First, we compute the gradients of L with respect to p_{i,y_i} in Eq. (1.25) as

$$\frac{\partial L}{\partial p_{i,y_i}} = -\frac{1}{N p_{i,y_i}}. \tag{1.28}$$

Note that

$$\frac{\partial L}{\partial p_{i,c}} = 0, \forall c \neq y_i. \tag{1.29}$$

Next, we compute

$$\begin{aligned}\frac{\partial L}{\partial z_{i,c}} &= \sum_{i'=1}^{N}\sum_{c'=1}^{C}\frac{\partial L}{\partial p_{i',c'}}\cdot\frac{\partial p_{i',c'}}{\partial z_{i,c}} \\ &= \sum_{c'=1}^{C}\frac{\partial L}{\partial p_{i,c'}}\cdot\frac{\partial p_{i,c'}}{\partial z_{i,c}} \\ &= \frac{\partial L}{\partial p_{i,y_i}}\cdot\frac{\partial p_{i,y_i}}{\partial z_{i,c}}.\end{aligned} \tag{1.30}$$

From the lecture, we know that

$$\frac{\partial p_{i,y_i}}{\partial z_{i,c}} = p_{i,y_i}(\delta_{y_i,c} - p_{i,c}), \tag{1.31}$$

where

$$\delta_{y_i,c} = \begin{cases} 1, & \text{if } c = y_i \\ 0, & \text{otherwise.} \end{cases} \tag{1.32}$$

By substituting Eq. (1.28) and Eq. (1.31) into Eq. (1.30), we have

$$\frac{\partial L}{\partial z_{i,c}} = -\frac{1}{N p_{i,y_i}}\cdot p_{i,y_i}(\delta_{y_i,c} - p_{i,c}) \tag{1.33}$$

$$= \frac{p_{i,c} - \delta_{y_i,c}}{N} \tag{1.34}$$

Lastly, we have $z_{i,c} = x_{i,c} - M_i$. It can be shown that this max-subtraction does not change the downstream gradients. Therefore, we have

$$\frac{\partial L}{\partial x_{i,c}} = \frac{\partial L}{\partial z_{i,c}} = \frac{p_{i,c} - \delta_{y_i,c}}{N}. \tag{1.35}$$

(1.4) L2 regularization. Implement the l2_regularization() function in neuralnet/layers.py after reading the following material.

L2 regularization implements the L2 regularization loss on the parameters in the network:

$$L(W) = \frac{\lambda}{2}\|W\|^2 = \frac{\lambda}{2}\sum_i W_i^2, \tag{1.36}$$

where the sum ranges over all scalar elements of the weight matrix W and λ is a hyperparameter controlling the regularization strength. The downstream gradients of this loss function is

$$\frac{\partial L}{\partial W_i} = \lambda W_i, \tag{1.37}$$

for each element in W.

Problem 2

(2 points) Implement a Two-Layer Network.

In this problem, you need to implement a two-layer network using the layers from problem 1. This network has two FC layers with one ReLU activation layer: input -> FC layer -> ReLU layer -> FC layer -> scores.

Complete the implementation of the TwoLayerNet class in neuralnet/two_layer_net.py.

First, you can see that the TwoLayerNet class is a subclass of the Classifer class defined in neuralnet/classifier.py. The Classifier class is a base class for image classification models. You do not need to implement anything in this class, but you should read through it to familiarize yourself with the API.

Second, in the neuralnet/linear_classifier.py script, a LinearClassifer class is implemented. You can study this implementation to see how these layers are used.

Finally, you can implement the TwoLayerNet class in neuralnet/two_layer_net.py. Your implementations for the forward and backward methods should use the modular forward and backward functions in these layers that you implemented in Problem 1.

After completing your implementation, you can run the script neuralnet/gradcheck_classifier.py to perform numeric gradient checking on both the linear classifier as well as the two-layer network you implemented. You should see errors less than 10^{-10} for the gradients of all parameters.

Problem 3

(2 points) Training a Two-Layer Network.

In this problem, you need to train your implemented two-layer network on the CIFAR-10 dataset for image classification. This dataset consists of 32 × 32 RGB images of 10 different categories. It provides 50,000 training images and 10,000 test images. Figure 2 shows a few example images from the dataset.

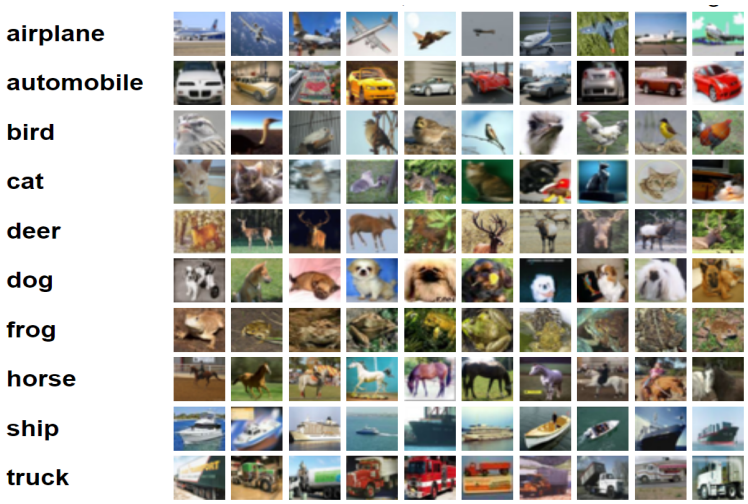

Figure 2: Example images for the CIFAR-10 dataset.

First, You need to use the script nerualnet/download_cifar.sh to download and unpack the CIFAR10 dataset. After downloading the dataset, implement the training_step() function in train.py.

The main function in train.py first set a few hyperparameters. Then it set up data samplers for the training set and the validation set of the CIFRA10 dataset. After initializing the two-layer network and the optimizer using stochastic gradient descent, it implements a training loop for training.

In this training loop, the training_step() function is called for each mini-batch. This function inputs the model, a minibatch of data, and the regularization strength. It computes a forward and backward pass through the model and returns both the loss and the gradient of the loss with respect to the model parameters. The loss should be the sum of two terms:

- A data loss term, which is the softmax loss between the model's predicted scores and the ground-truth image labels.
- A regularization loss term, which penalizes the L2 norm of the weight matrices of all the fully-connected layers of the model. You should not apply L2 regularization to the biases.

Figure 4 shows an examples for linear regression with a L2 loss. You can follow this example to compute the loss for the two-layer network and the gradients of the loss with respect to the parameters of the network in the training_step() function.

After your implementation, it is time to train the network. Run the script neuralnet/train.py to train a two-layer network on the CIFAR10 dataset. The script will print out training losses and

```python
from layers import fc_forward, fc_backward, l2_loss

def linear_regression_step(X, y, W, b):
    y_pred, cache = fc_forward(X, W, b)
    loss, grad_y_pred = l2_loss(y_pred, y)
    grad_X, grad_W, grad_b = fc_backward(grad_y_pred, cache)
    return grad_W, grad_b
```

Figure 3: Linear regression with a L2 loss function.

train and val set accuracy as it trains. After training concludes, the script will also make a plot of the training losses as well as the training and validation-set accuracy of the model during training. By default this will be saved in a file plot.pdf, but this can be customized with the flag $--plot-file$. You should see a plot that looks like this:

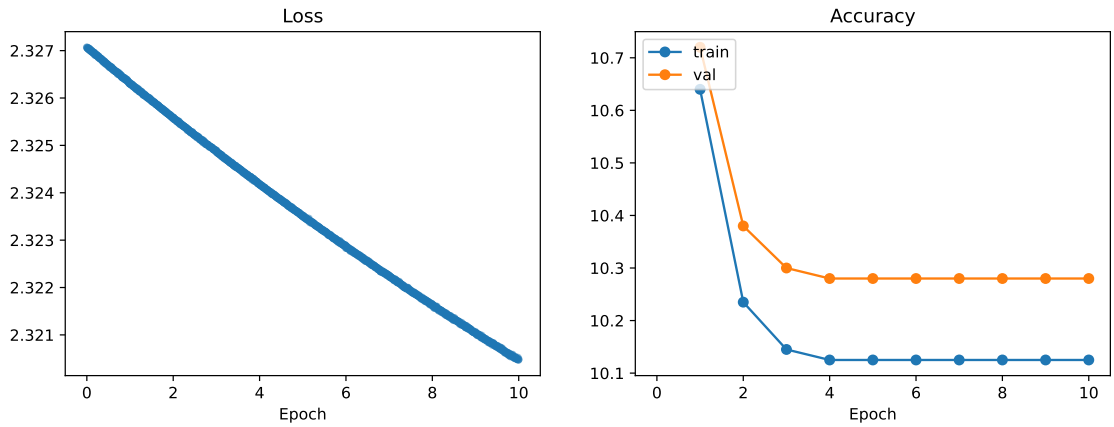

Figure 4: Trainng loss and train-validation accuracy.

Unfortunately, it seems that your model is not training very effectively – the training loss has not decreased much from its initial value of 2.3, and the training and validation accuracies are very close to 10% which is what we would expect from a model that randomly guesses a category label for each input.

You will need to tune the hyperparameters of your model in order to improve it. Try changing the hyperparameters of the model in the provided space of the main function of neuralnet/train.py. You can consider changing any of the following hyperparameters:

- num_train: The number of images to use for training
- hidden_dim: The width of the hidden layer of the model
- batch_size: The number of examples to use in each minibatch during SGD
- num_epochs: How long to train. An epoch is a single pass through the training set.
- learning_rate: The learning rate to use for SGD

- reg: The strength of the L2 regularization term

You should tune the hyperparameters and train a model that achieves at least 40% on the validation set. In your homework submission, include the loss / accuracy plot for your best model. After tuning your model, run your best model exactly once on the test set using the script neuralnet/test.py.

You may not need to change all of the hyperparameters. Some are fine at their default values. Your model should not take an excessive amount of time to train. For reference, our hyperparameter settings achieve 42% accuracy on the validation set in less than 1 minute of training on a desktop with an Intel i9 CPU.