Prof. Fabio Galasso <galasso@di.uniroma1.it>

Exercise 1: Deep Neural Network and Backpropagation

Deep neural networks have shown staggering performances in various learning tasks, including computer vision, natural language processing, and sound processing. They have made the model designing more flexible by enabling end-to-end training.

In this exercise, we get to have a first hands-on experience with neural network training. Many frameworks (e.g. PyTorch, Tensorflow, Caffe) allow easy usage of deep neural networks without precise knowledge on the inner workings of backpropagation and gradient descent algorithms. While these are very useful tools, it is important to get a good understanding of how to implement basic network training from scratch, before using this libraries to speed up the process. For this purpose we will implement a simple two-layer neural network and its training algorithm based on back-propagation using only basic matrix operations in questions 1 to 3. In question 4, we will use a popular deep learning library, PyTorch, to do the same and understand the advantages offered in using such tools.

As a benchmark to test our models, we consider an image classification task using the widely used CIFAR-10 dataset. This dataset consists of 50000 training images of 32x32 resolution with 10 object classes, namely airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The task is to code and train a parameterised model for classifying those images. This involves

- Implementing the feedforward model (Question 1).
- Implementing the backpropagation algorithm (gradient computation) (Question 2).
- Training the model using stochastic gradient descent and improving the model training with better hyper-parameters (Question 3).
- Using the PyTorch Library to implement the above and experiment with deeper networks (Question 4).

A note on notation: throughout the exercise, notation v_i is used to denote the i^{th} element of vector v

Questions 1-3 are based on the script `ex2_FCnet.py` and question 4 is based on the script `ex2_pytorch.py`. To download the CIFAR-10 dataset please execute the script `datasets/get_datasets.sh` or use the function `torchvision.datasets.CIFAR10`, as illustrated in the section “Load the CIFAR-10 dataset” in the file `ex2_pytorch.py`.

This exercise is provided for practicing basic concepts of Deep Neural Networks, which are important for the understanding of complex ones. Completing this exercise is not compulsory but recommended.

Question 1: Implementing the feedforward model

In this question we will implement a two-layered a neural network architecture as well as the loss function to train it. Starting from the main file `ex2_FCnet.py`, complete the required code in the `two_layernet.py` to complete this question. Refer to the comments in the code to the exact places where you need to fill in the code.

Model architecture. Our architecture is shown in 1. It has an input layer, and two model layers – a hidden and an output layer. We start with randomly generated toy inputs of 4 dimensions and number of classes $K = 3$ to build our model in Questions 1, 2 and 3. Then we would use images from CIFAR-10 dataset to test our model on a real-world task in Question 3. Hence input layer is 4 dimensional for now.

In the hidden layer, there are 10 units. The input layer and the hidden layer are connected via linear weighting matrix $W^{(1)} \in \mathbb{R}^{10x4}$ and the bias term $b^{(1)} \in \mathbb{R}^{10}$. The parameters $W^{(1)}$ and $b^{(1)}$ are to be learnt later on. A linear operation is performed, $W^{(1)}x + b^{(1)}$, resulting in a 10-dimensional vector $z^{(2)}$. It is then followed by a ReLU non-linear activation ϕ, applied element-wise on each unit, resulting in the activations $a^{(2)} = \phi(z^{(2)})$. The ReLU function has the following form:

$$\phi(u) = \begin{cases} u, & \text{if } u \geq 0 \\ 0, & \text{if } u < 0 \end{cases} \tag{1}$$

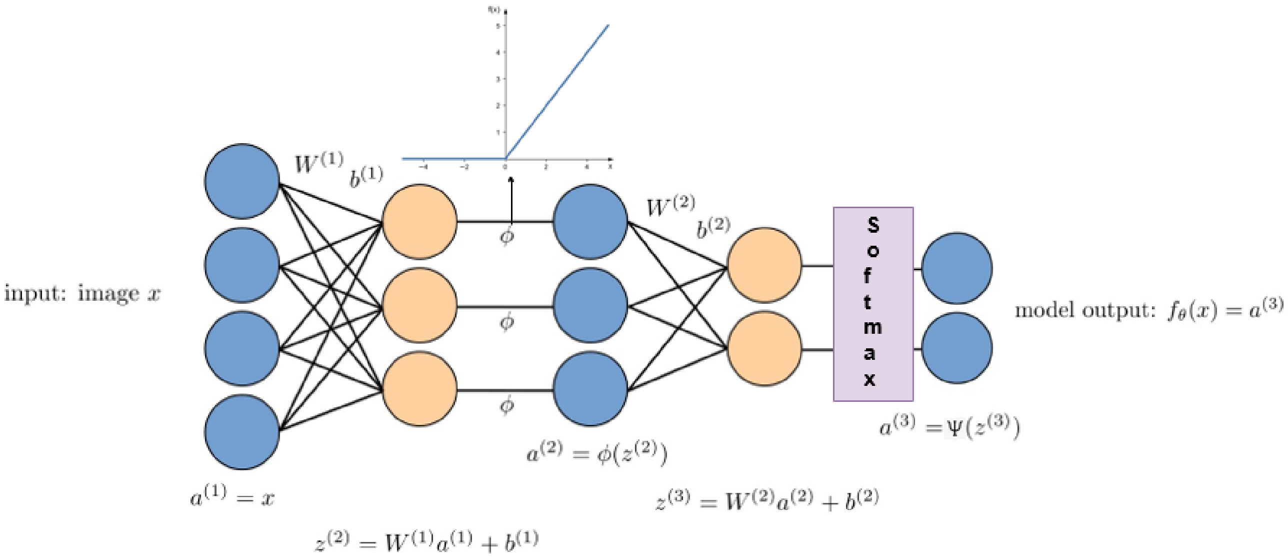
input: image x
$W^{(1)}$
$b^{(1)}$
ϕ
ϕ
ϕ
$W^{(2)}$
$b^{(2)}$
S o f t m a x
model output: $f_\theta(x) = a^{(3)}$
$a^{(3)} = \Psi(z^{(3)})$
$a^{(2)} = \phi(z^{(2)})$
$a^{(1)} = x$
$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$
$z^{(2)} = W^{(1)}a^{(1)} + b^{(1)}$

Figure 1: Visualisation of the two layer fully connected network, used in Q1-Q3.

A similar linear operation is performed on $a^{(2)}$, resulting in $z^{(3)} = W^{(2)}a^{(2)}$, where $W^{(2)} \in \mathbb{R}^{3x10}$ and $b^{(2)} \in \mathbb{R}^3$; it is followed by the softmax activation to result in $a^{(3)} = \psi(z^{(3)})$. The softmax function is defined as

$$\psi(u_i) = \frac{e^{u_i}}{\sum_j e^{u_j}} \tag{2}$$

The final functional form of our model is thus defined by

$$\begin{aligned} a^{(1)} &= x \\ z^{(2)} &= W^{(1)}a^{(1)} + b^{(1)} \\ a^{(2)} &= \phi\left(z^{(2)}\right) \\ z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\ f_\theta(x) := a^{(3)} &= \psi\left(z^{(3)}\right) \end{aligned} \tag{3}$$

which takes a flattened 4 dimensional vector as input and outputs a 3-dimensional vector, each entry in the output $f_\theta(x)_k$ representing the probability of image x corresponding to the class k. We indicate all the network parameters by $\theta = \left(W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}\right)$.

Implementation. We are now ready to implement the feedforward neural network.

a) Implement the code in `two_layernet.py` for the feedforward model. You are required to implement (3). Verify that the scores you generate for the toy inputs match the correct scores given in the `ex2_FCnet.py`.

b) We later guide the neural network parameters $\theta = \left(W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}\right)$ to fit to the given data and label pairs. We do so by minimising the loss function. A popular choice of the loss function for training neural network for multi-class classification is the cross-entropy loss. For a single input sample x_i, with label y_i, the loss function is defined as :

$$\begin{aligned} J(\theta, x_i, y_i) &= -\log P\left(Y = y_i, X = x_i\right) \\ &= -\log f_\theta\left(x_i\right)_{y_i} \\ &= -\log \psi\left(z_i^{(3)}\right)_{y_i} \\ J\left(\theta, x_i, y_i\right) &= -\log\left[\frac{\exp\left(z_i^{(3)}\right)_{y_i}}{\sum_{j=1}^{K}\exp\left(z_i^{(3)}\right)_j}\right] \end{aligned} \tag{4}$$

Averaging over the whole training set, we get

$$J\left(\theta, \{x_i, y_i\}_{i=1}^{N}\right) = \frac{1}{N}\sum_{i=1}^{N} -\log\left[\frac{\exp\left(z_i^{(3)}\right)_{y_i}}{\sum_{j=1}^{K}\exp\left(z_i^{(3)}\right)_j}\right] \tag{5}$$

where K is the number of classes and v_j stands for the j-th element of the vector v. Note that if the model has perfectly fitted to the data (i.e. $f_\theta(x_i)_k = 1$ whenever x_i belongs to class k and 0 otherwise), then J attains the minimum of 0.

Apart from trying to correctly predict the label, we have to prevent overfitting the model to the current training data. This is done by encoding our prior belief that the correct model should be simple (Occam's razor); we add an L_2 regularisation term over the model parameters θ. Specifically, the loss function is defined by:

$$\tilde{J}\left(\theta, \{x_i, y_i\}_{i=1}^N\right) = \frac{1}{N}\sum_{i=1}^{N} -\log\left[\frac{\exp\left(z_i^{(3)}\right)_{y_i}}{\sum_{j=1}^{K}\exp\left(z_i^{(3)}\right)_j}\right] + \lambda\left(\left\|W^{(1)}\right\|_2^2 + \left\|W^{(2)}\right\|_2^2\right) \tag{6}$$

where $||.||_2^2$ is the squared L_2 norm. For example,

$$\left\|W^{(1)}\right\|_2^2 = \sum_{p=1}^{10}\sum_{q=1}^{4}\left(W_{pq}^{(1)}\right)^2 \tag{7}$$

By changing the value of λ it is possible to give weights to your prior belief on the degree of simplicity (regularity) of the true model.

Implement the final loss function in `two_layernet.py` and let it return the loss value. Verify the code by running and matching the output cost 1.30378789133.

c) To be able to train the above model on large datasets, with larger layer widths, the code has to be very efficient. To do this you should avoid using any python for loops in the forward pass and instead use matrix and vector multiplication routines in the Numpy library. If you have written the code of parts (a) and (b) using loops, convert it to vectorized version using NumPy operations.

Question 2: Backpropagation

We train the model by solving

$$\min_{\theta} \tilde{J}\left(\theta, \{x_i, y_i\}_{i=1}^N\right) \tag{8}$$

via stochastic gradient descent. We therefore need an efficient computation of the gradients $\nabla_\theta \tilde{J}$. We use backpropagation of top layer error signals to the parameters θ at different layers.

In this question, you will be required to implement the backpropagation algorithm yourself from a pseudocode. We will give a high-level description of what is happening at each line.

We will exercise on the derivation of backpropagation algorithm. A prior knowledge on standard vector calculus including the chain rule would be helpful.

Backpropagation. The backpropagation algorithm is simply a sequential application of chain rule. It is applicable to any (sub-) differentiable model that is a composition of simple building blocks. In this exercise, we focus on the architecture with stacked layers of linear transformation + ReLU non-linear activation.

The intuition behind the backpropagation algorithm is as follows. Given a training example (x_i, y_i), we first run the feedforward to compute all the activations throughout the network, including the output value of the model $f_\theta(x_i)$ and the loss $\tilde{J}$. Then, for each parameter in the model we want to compute the effect that parameter has on the loss. This is done by computing the derivatives of the loss w.r.t. each model parameter.

The backpropagation algorithm is performed from the top of the network (loss layer) towards the bottom. It sequentially computes the gradient of the loss function with respect to each layer activations and parameters.

Let's start by deriving the gradients of the un-regularized loss function J w.r.t. the final layer activations $z_i^{(3)}$. We will then use this in the chain rule to compute analytical expressions for gradients of all the model parameters.

a) Verify that the loss function defined in Eq. (5) has gradient w.r.t. $z_i^{(3)}$ as below:

$$\frac{\partial J}{\partial z_i^{(3)}}\left(\theta, \{x_i, y_i\}_{i=1}^N\right) = \frac{1}{N}\left(\psi\left(z_i^{(3)}\right) - \Delta_i\right) \tag{9}$$

where Δ_i is a vector of K dimensions with

$$(\Delta_i)_j = \begin{cases} 1, & \text{if } j = y_i \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

Note: in these derivations we have defined $z = Wx + b$, which mismatches what explained in the lectures, where we instead have adopted $z = xW + b$. You are suggested to adopt the second formulation in the code, to match the lectures. This second approach is particularly convenient when considering batches of N samples and implementing the sample gradient accumulation by matrix multiplication (instead of using a summation / for loop, as shown above).

b) To compute the effect of the weight matrix $W^{(2)}$ on the loss in Eq. (5) incurred by the network, we compute the partial derivative of the loss function with respect to $W^{(2)}$. This is done by applying the chain rule. Verify that the partial derivative of the loss w.r.t. $W^{(2)}$ is

$$\begin{aligned}\frac{\partial J}{\partial W^{(2)}}\left(\theta, \{x_i, y_i\}_{i=1}^N\right) &= \sum_{i=1}^N \frac{\partial J}{\partial z_i^{(3)}} \cdot \frac{\partial z_i^{(3)}}{\partial W^{(2)}} \\ &= \sum_{i=1}^N \frac{1}{N}\left(\psi\left(z_i^{(3)}\right) - \Delta_i\right) a_i^{(2)^T}\end{aligned} \tag{11}$$

Similarly, verify that the regularized loss in Eq. (6) has the derivative:

$$\frac{\partial \tilde{J}}{\partial W^{(2)}} = \sum_{i=1}^N \frac{1}{N}\left(\psi\left(z_i^{(3)}\right) - \Delta_i\right) a_i^{(2)^T} + 2\lambda W^{(2)} \tag{12}$$

You may refer to the formulae of matrix backpropagation seen in the class: given $Y = XW$, then $\frac{\delta L}{\delta X} = \frac{\delta L}{\delta Y} W^T$ and $\frac{\delta L}{\delta W} = X^T \frac{\delta L}{\delta Y}$. But you should watch out for the different formulation in Eq. (3), as also pointed out above.

c) We can repeatedly apply chain rule as discussed above to obtain the derivatives of the loss with respect to all the parameters of the model $\theta = \left(W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}\right)$. Derive the expressions for the derivatives of the regularized loss in Eq. (6) w.r.t. $W^{(1)}$, $b^{(1)}$, $b^{(2)}$ now.

d) Using the expressions you obtained for the derivatives of the loss w.r.t. the model parameters, implement the backpropagation algorithm in the file `two_layernet.py`. Run the `ex2_FCnet.py` and verify that the gradients you obtained are correct using numerical gradients (already implemented in the code). The maximum relative error between the gradients you compute and the numerical gradients should be less than 1e-8 for all parameters.

Question 3: Stochastic gradient descent training

We have implemented the backpropagation algorithm for computing the parameter gradients and have verified that it indeed gives the correct gradient. We are now ready to train the network. We solve Eq. (8) with the stochastic gradient descent.

Stochastic gradient descent (SGD). Typically neural networks are large and are trained with millions of data points. It is thus often infeasible to compute the gradient $\nabla_\theta \tilde{J}$ that requires the accumulation of the gradient over the entire training set. Stochastic gradient descent addresses this problem by simply accumulating the gradient over a small random subset of the training samples (mini-batch) at each iteration. Specifically, the algorithm is as follows

```
Data: Training data $\{(x_i, y_i)\}_{i=1,\dots,N}$ , initial network parameter $\theta^{(0)}$, regularisation hyper-parameter $\lambda$,
      learning rate $\alpha$, batch size $B$, iteration limit $T$
Result: Trained parameter $\theta^{(T)}$
for $t = 1, \dots, T$ do
    $\left\{(x'_j, y'_j)\right\}_{i=1}^B \leftarrow$ a random subset of the original training set $\{(x_i, y_i)\}_{i=1}^N$ ;
    $v \leftarrow -\alpha \nabla_\theta \tilde{J}\left(\theta^{(t-1)}, \left\{(x'_j, y'_j)\right\}_{i=1}^B\right)$ ;
    $\theta^{(t)} \leftarrow \theta^{(t-1)} + v$ ;
end
```

Algorithm 1: Stochastic gradient descent with momentum

where the gradient $\nabla_\theta \tilde{J}\left(\theta^{(t-1)}, \left\{(x'_j, y'_j)\right\}_{i=1}^B\right)$ is computed only on the current randomly sampled batch. Intuitively, $v = -\nabla_\theta \tilde{J}\left(\theta^{(t-1)}\right)$ gives the direction to which the loss $\tilde{J}$ decreases the most (locally), and therefore we follow that direction by updating the parameters towards that direction $\theta^{(t)} = \theta^{(t-1)} + v$.

a) Implement the stochastic gradient descent algorithm in `two_layernet.py` and run the training on the toy data. Your model should be able to obtain loss = 0.02 on the training set and the training curve should look similar to the one shown in Fig. 2.

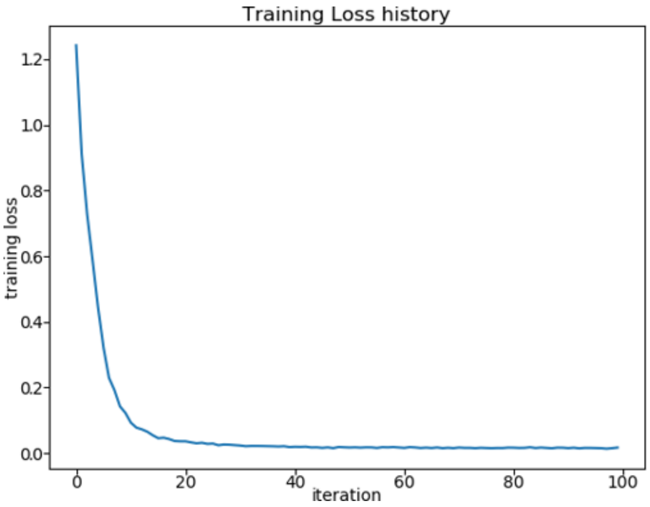

Figure 2: Example training curve on the toy dataset.

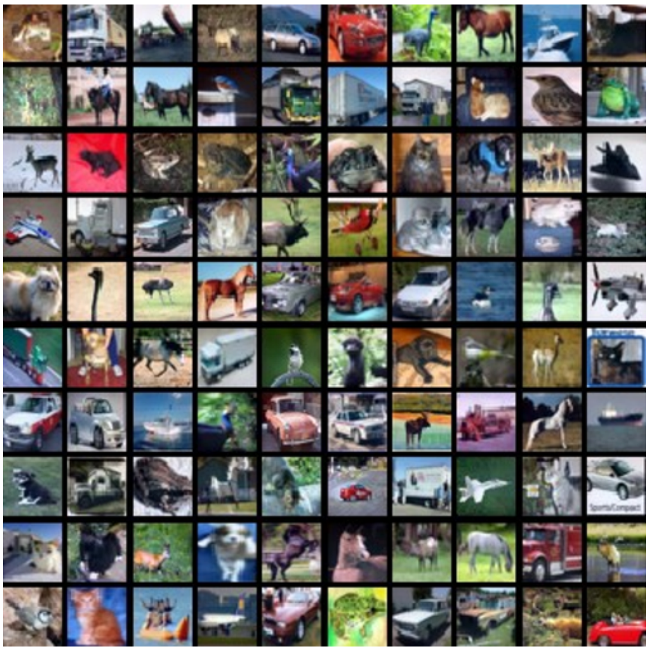

Figure 3: Example images from the CIFAR-10 dataset.

b) We are now ready to train our model on real image dataset. For this we will use the CIFAR-10 dataset. Since the images are of size 32 × 32 pixels with 3 color channels, this gives us 3072 input layer units, represented by a vector $x \in \mathbb{R}^{3072}$. See Fig. 3 for example images from the dataset. The code to load the data and train the model is provided with some default hyper-parameters in `ex2_FCnet.py`. With default hyper-parametres, if previous questions have been done correctly, you should get validation set accuracy of about 29%. This is very poor. Your task is to debug the model training and come up with better hyper-parameters to improve the performance on the validation set. Visualize the training and validation performance curves to help with this analysis. There are several pointers provided in the comments in the `ex2_FCnet.py` to help you understand why the network might be underperforming (Lines 232–259). Once you have tuned your hyper-parameters, and get validation accuracy greater than 48% run your best model on the test set once and report the performance.

Question 4: Implement multi-layer perceptron using PyTorch library

So far we have implemented a two-layer network by explicitly writing down the expressions for forward, backward computations and training algorithms using simple matrix multiplication primitives from the NumPy library.

However there are many libraries available designed make experimenting with neural networks faster, by abstracting away the details into re-usable modules. One such popular open-source library is PyTorch (`https:`

//pytorch.org/). In this final question we will use PyTorch library to implement the same two-layer network we did before and train it on the CIFAR-10 dataset. However, extending a two-layer network to a three or four layered one is a matter of changing two-three lines of code using PyTorch. We will take advantage of this to experiment with deeper networks to improve the performance on the CIFAR-10 classification. This question is based on the file ex2_pytorch.py.

To install the PyTorch library follow the instruction in https://pytorch.org/get-started/locally/ . If you have access to a Graphics Processing Unit (GPU), you can install the GPU version and run the exercise on GPU for faster run times. If not, you can install the cpu version (select CUDA version None) and run on the cpu. Having GPU access is not necessary to complete the exercise. There are good tutorials for getting started with PyTorch on their website (https://pytorch.org/tutorials/).

a) Complete the code to implement a multi-layer perceptron network in the class *MultiLayerPerceptron* in ex2_pytorch.py. This includes instantiating the required layers from torch.nn and writing the code for forward pass. Initially you should write the code for the same two-layer network we have seen before.

b) Complete the code to train the network. Make use of the loss function torch.nn.CrossEntropyLoss to compute the loss and loss.backward() to compute the gradients. Once gradients are computed, optimizer.step() can be invoked to update the model. Your should be able to achieve similar performance ($> 48\%$ accuracy on the validation set) as in Q3. Report the final validation accuracy you achieve with a two-layer network.

c) Now that you can train the two layer network to achieve reasonable performance, try increasing the network depth to see if you can improve the performance. Experiment with networks of at least 2, 3, 4, and 5 layers, of your chosen configuration. Report the training and validation accuracies for these models and discuss your observations. Run the evaluation on the test set with your best model and report the test accuracy.

Exercise credits: Prof. Bernt Schiele, Prof. Mario Fritz, Rakshith Shetty, Yang He, Dr. Seong Joon Oh.