16-720 Computer Vision: Homework 5 (Spring 2022) Neural Networks for Recognition

Instructor: David Held
TAs: Aditya Ghuge, Jack Good, Jenny Nan, Rohan Chacko
Due: Wednesday April 13, 2022 23:59:59

Instructions

1. **Integrity and collaboration:** Students are encouraged to work in groups but each student must submit their own work. Include the names of your collaborators in your write up. Code should **NOT** be shared or copied. Please **DO NOT** use external code unless permitted. Plagiarism is prohibited and may lead to failure of this course.

2. **Questions:** If you have any question, please look at piazza first.

3. **Submission:** The submission is on Gradescope, **you will be submitting both your writeup and code zip file**. The zip file, `<andrew-id.zip>` contains your code and any results files we ask you to save. **Note: You have to submit your writeup separately to Gradescope, and include results (and code snippets, when requested) in the write-up**.

4. **Do not** submit anything from the `data/` folder in your submission.

5. For your code submission, **do not** use any libraries other than *numpy, scipy, scikit-image, matplotlib* and (in the appropriate section) *pytorch*. Including other libraries (for example, cv2, ipdb, etc.) **may lead to loss of credit** on the assignment.

6. To get the data, we have included some scripts in `scripts`.

7. For each coding question, refer to the comments inside the given python scripts to see which function to implement. Insert your code into the designated place (where it says `your code here`), and **DO NOT** remove any of the given code elsewhere.

Contents

1   Theory

Q1.1 Theory   [**3 points**] Prove that softmax is invariant to translation, that is

$$softmax(x) = softmax(x + c) \quad \forall c \in \mathbb{R}.$$

Softmax is defined as below, for each index i in a vector x.

$$softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Often we use $c = -\max x_i$. Why is that a good idea? (Tip: consider the range of values that numerator will have with $c = 0$ and $c = -\max x_i$)

Q1.2 Theory   [**3 points**] Softmax can be written as a three step processes, with $s_i = e^{x_i}$, $S = \sum s_i$ and $softmax(x_i) = \frac{1}{S}s_i$.

- As $x \in \mathbb{R}^d$, what are the properties of $softmax(x)$, namely what is the range of each element? What is the sum over all elements?

- One could say that *"softmax takes an arbitrary real valued vector x and turns it into a _____"*.

- Can you see the role of each step in the multi-step process now? Explain them.

Q1.3 Theory   [**3 points**] Show that multi-layer neural networks without a non-linear activation function are equivalent to linear regression.

Q1.4 Theory   [**3 points**] Given the sigmoid activation function $\sigma(x) = \frac{1}{1+e^{-x}}$, derive the gradient of the sigmoid function and show that it can be written as a function of $\sigma(x)$ (without having access to x directly)

Q1.5 Theory **[12 points]** Given $y = Wx + b$ (or $y_i = \sum_{j=1}^{d} x_j W_{ij} + b_i$), and the gradient of some loss J with respect y, show how to get $\frac{\partial J}{\partial W}$, $\frac{\partial J}{\partial x}$ and $\frac{\partial J}{\partial b}$. Be sure to do the derivatives with scalars and re-form the matrix form afterwards. Here is some notional suggestions.

$$\frac{\partial J}{\partial y} = \delta \in \mathbb{R}^{k \times 1} \quad W \in \mathbb{R}^{k \times d} \quad x \in \mathbb{R}^{d \times 1} \quad b \in \mathbb{R}^{k \times 1}$$

We won't grade the derivative with respect to b but you should do it anyways, you will need it later in the assignment.

Q1.6 Theory [4 points] When the neural network applies the elementwise activation function (such as sigmoid), the gradient of the activation function scales the backpropogation update. This is directly from the chain rule, $\frac{d}{dx} f(g(x)) = f'(g(x))g'(x)$.

1. Consider the sigmoid activation function for deep neural networks. Why might it lead to a "vanishing gradient" problem if it is used for many layers (consider plotting Q1.4)?

2. Often it is replaced with $\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$. What are the output ranges of both tanh and sigmoid? Why might we prefer tanh ?

3. Why does $\tanh(x)$ have less of a vanishing gradient problem? (plotting the derivatives helps! for reference: $\tanh'(x) = 1 - \tanh(x)^2$)

4. tanh is a scaled and shifted version of the sigmoid. Show how $\tanh(x)$ can be written in terms of $\sigma(x)$.

2 Implement a Fully Connected Network

When implementing the following functions, make sure you run `python/run_q2.py` along the way to check if your implemented functions work as expected.

2.1 Network Initialization

Q2.1.1 Theory [3 points] Why is it not a good idea to initialize a network with all zeros? If you imagine that every layer has weights and biases, what can a zero-initialized network output after training?

Q2.1.2 Code [3 points] In `python/nn.py`, implement a function to initialize one layer's weights with Xavier initialization [1], where $Var[w] = \frac{2}{n_{in}+n_{out}}$ where n is the dimensionality of the vectors and you use a **uniform distribution** to sample random numbers (see eq 16 in the paper). **Include your code in the writeup.**

Q2.1.3 Theory [2 points] Why do we initialize with random numbers? Why do we scale the initialization depending on layer size (see near Fig 6 in the paper)?

2.2 Forward Propagation

The appendix (sec 8) has the math for forward propagation, we will implement it here.

Q2.2.1 Code [4 points] In `python/nn.py`, implement sigmoid, along with forward propagation for a single layer with an activation function, namely $y = \sigma(XW + b)$, returning the output and intermediate results for an $N \times D$ dimension input X, with examples along the rows, data dimensions along the columns. **Include your code in the writeup.**

Q2.2.2 Code [3 points] In `python/nn.py`, implement the softmax function. Be sure to use the numerical stability trick you derived in Q1.1 softmax. **Include your code in the writeup.**

Q2.2.3 Code [3 points] In `python/nn.py`, write a function to compute the accuracy of a set of labels, along with the scalar loss across the data. The loss function generally used for classification is the cross-entropy loss.

$$L_{\mathbf{f}}(\mathbf{D}) = - \sum_{(\mathbf{x},\mathbf{y}) \in \mathbf{D}} \mathbf{y} \cdot \log(\mathbf{f}(\mathbf{x}))$$

Here $\mathbf{D}$ is the full training dataset of data samples $\mathbf{x}$ ($N \times 1$ vectors, N = dimensionality of data) and labels $\mathbf{y}$ ($C \times 1$ one-hot vectors, C = number of classes), and $\mathbf{f} : \mathbb{R}^N \to [0, 1]^C$ is the classifier. The log is natural log. **Include your code in the writeup.**

2.3 Backwards Propagation

Q2.3.1 Code [7 points] In `python/nn.py`, write a function to compute backpropogation for a single layer, given the original weights, the appropriate intermediate results, and given gradient with respect to the loss. You should return the gradient with respect to X so you can feed it into the next layer. As a size check, your gradients should be the same dimensions as the original objects. **Include your code in the writeup.**

2.4 Training Loop

You will tend to see gradient descent in three forms: "normal", "stochastic" and "batch". "Normal" gradient descent aggregates the updates for the entire dataset before changing the weights. Stochastic gradient descent applies updates after every single data example. Batch gradient descent is a compromise, where random subsets of the full dataset are evaluated before applying the gradient update.

Q2.4 Code [5 points] In `python/nn.py`, write a function that takes the entire dataset (`x` and `y`) as input and splits it into random batches. In `python/run_q2.py`, write a training loop that iterates over the batches, does forward and backward propagation, and applies a gradient update. The provided code samples batches only once, but it is also common to sample new random batches at each epoch. You may optionally try both strategies and note any difference in performance. **Include your code in the writeup.**

2.5 Numerical Gradient Checker

Q2.5 [5 points] In `python/run_q2.py`, implement a numerical gradient checker. Instead of using the analytical gradients computed from the chain rule, add ϵ offset to each element in

the weights, and compute the numerical gradient of the loss with central differences. Central differences is just $\frac{f(x+\epsilon)-f(x-\epsilon)}{2\epsilon}$. Remember, this needs to be done for each scalar dimension in all of your weights independently. This should help you check your gradient code, so there is no need to show the result, but do **include your code in the writeup.**

3   Training Models

First, from the scripts folder, run the script `get_data.sh`. This will download and unzip
http://www.cs.cmu.edu/~lkeselma/16720a_data/data.zip
http://www.cs.cmu.edu/~lkeselma/16720a_data/images.zip
and extract them to `data` and `image` folders. (For those who are on operating systems like Windows and cannot run the script, you can also manually download and unzip the data.)

Since our input images are 32×32 images, unrolled into one 1024 dimensional vector, that gets multiplied by $\mathbf{W}^{(1)}$, each row of $\mathbf{W}^{(1)}$ can be seen as a weight image. Reshaping each row into a 32×32 image can give us an idea of what types of images each unit in the hidden layer has a high response to.

We have provided you three data `.mat` files to use for this section. The training data in `nist36_train.mat` contains samples for each of the 26 upper-case letters of the alphabet and the 10 digits. This is the set you should use for training your network. The cross-validation set in `nist36_valid.mat` contains samples from each class, and should be used in the training loop to see how the network is performing on data that it is not training on. This will help to spot overfitting. Finally, the test data in `nist36_test.mat` contains testing data, and should be used for the final evaluation on your best model to see how well it will generalize to new unseen data.

Use `python/run_q3.py` for this question, and refer to the comments for what to implement.

Q3.1 Code [5 points]   Train a network from scratch. Use a single hidden layer with 64 hidden units, and train for at least 50 epochs. The script will generate two plots: one showing the accuracy on both the training and validation set over the epochs, and the other showing the cross-entropy loss averaged over the data. Tune the batch size and learning rate to get an accuracy on the validation set of at least 75%. Include the plots in your writeup.

Q3.2 Writeup [3 points]   Use the script to train three networks, one with your tuned learning rate, one with 10 times that learning rate, and one with one tenth that learning rate. Include all six plots in your writeup (two will be the same from the previous question). Comment on how the learning rates affect the training, and report the final accuracy of the best network on the test set.

Q3.3 Writeup [3 points]   The script will visualize the first layer weights as 64 32x32 images, both immediately after initialization and after fully training. Include both visualizations in your writeup. Comment on the learned weights and compare them to the initialized weights. Do you notice any patterns?

Q3.4 Writeup [3 points]   Visualize the confusion matrix of the test data for your best model. Comment on the top few pairs of classes that are most commonly confused.
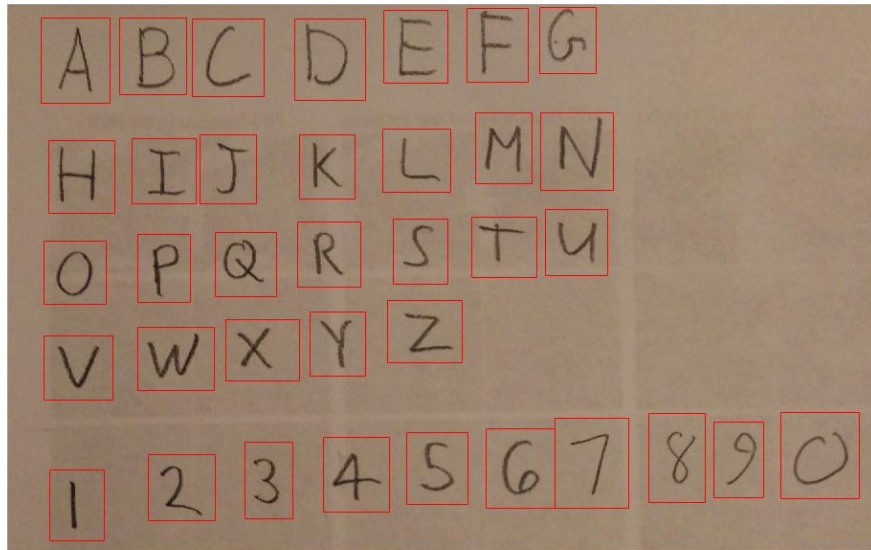
4   Extract Text from Images

Figure 1: Sample image with handwritten characters annotated with boxes around each character.

Now that you have a network that can recognize handwritten letters with reasonable accuracy, you can now use it to parse text in an image. Given an image with some text on it, our goal is to have a function that returns the actual text in the image. However, since your neural network expects a a binary image with a single character, you will need to process the input image to extract each character. There are various approaches that can be done so feel free to use any strategy you like.

Here we outline one possible method, another is that given in a tutorial

1. Process the image (blur, threshold, opening morphology, etc., perhaps in that order) to classify all pixels as being part of a character or background.

2. Find connected groups of character pixels (see skimage.measure.label). Place a bounding box around each connected component.

3. Group the letters based on which line of the text they are a part of, and sort each group so that the letters are in the order they appear on the page.

4. Take each bounding box one at a time and resize it to 32×32, classify it with your network, and report the characters in order (inserting spaces when it makes sense).

Since the network you trained likely does not have perfect accuracy, you can expect there to be some errors in your final text parsing. Whichever method you choose to implement for the character detection, you should be able to place a box on most of the characters in the image. We have provided you with 01_list.jpg, 02_letters.jpg, 03_haiku.jpg and 04_deep.jpg to test your implementation on.

Q4.1 Theory [3 points] The method outlined above is pretty simplistic, and while it works for the given text samples, it makes several assumptions. What are two big assumptions that the sample method makes? In your writeup, include two example images where you expect the character detection to fail (either miss valid letters, or respond to non-letters).

Q4.2 Code [10 points] In `python/q4.py`, implement the function to find letters in the image. Given an RGB image, this function should return bounding boxes for all of the located handwritten characters in the image, as well as a binary black-and-white version of the image `im`. Each row of the matrix should contain `[y1,x1,y2,x2]` the positions of the top-left and bottom-right corners of the box. The black and white image should be floating point, 0 to 1, with the characters in black and background in white. **Include your code in the writeup.**

Q4.3 Writeup [5 points] Using `python/run_q4.py`, visualize all of the located boxes on top of the binary image to show the accuracy of your `findLetters(..)` function. Include all the resulting images in your writeup.

Q4.4 Code/Writeup [8 points] In `python/run_q4.py`, you will now load the image, find the character locations, classify each one with the network you trained in **Q3.1**, and return the text contained in the image. Be sure you try to make your detected images look like the images from the training set. Visualize them and act accordingly.

Run your `run_q4` on all of the provided sample images in `images/`. Include the extracted text in your writeup. It is fine if your code ignores spaces, but if so, please add them manually in the writeup.

5   Image Compression with Autoencoders

An autoencoder is a neural network that is trained to attempt to copy its input to its output, but it usually allows copying only approximately. This is typically achieved by restricting the number of hidden nodes inside the autoencoder; in other words, the autoencoder would be forced to learn to *represent* data with this limited number of hidden nodes. This is a useful way of learning compressed representations. In this section, we will continue using the NIST36 dataset you have from the previous questions. Use `python/run_q5.py` for this question.

5.1   Building the Autoencoder

Q5.1.1 Code [5 points] Due to the difficulty in training auto-encoders, we have to move to the $relu(x) = max(x, 0)$ activation function. It is provided for you in `util.py`. Implement an autoencoder where the layers are

- 1024 to 32 dimensions, followed by a ReLU

- 32 to 32 dimensions, followed by a ReLU

- 32 to 32 dimensions, followed by a ReLU

- 32 to 1024 dimensions, followed by a sigmoid (this normalizes the image output for us)

The loss function that you're using is total squared error for the output image compared to the input image (they should be the same!). **Include your code in the writeup.**

Q5.1.2 Code [5 points]   To help even more with convergence speed, we will implement momentum. Now, instead of updating $W = W - \alpha \frac{\partial J}{\partial W}$, we will use the update rules $M_W = 0.9 M_W - \alpha \frac{\partial J}{\partial W}$ and $W = W + M_W$. To implement this, populate the parameters dictionary with zero-initialized momentum accumulators, one for each parameter. Then simply perform both update equations for every batch. **Include your code in the writeup.**

5.2   Training the Autoencoder

Q5.2 Writeup/Code [5 points]   Using the provided default settings, train the network for 100 epochs. Plot the training loss curve and include it in the writeup. What do you observe?

5.3   Evaluating the Autoencoder

Q5.3.1 Writeup/Code [5 points]   Now let's evaluate how well the autoencoder has been trained. Select 5 classes from the total 36 classes in the validation set and for each selected class include in your report 2 validation images and their reconstruction. What differences do you observe in the reconstructed validation images compared to the original ones?

Q5.3.2 Writeup [5 points]   Let's evaluate the reconstruction quality using Peak Signal-to-noise Ratio (PSNR). PSNR is defined as

$$\mathrm{PSNR} = 20 \times \log_{10}(\mathrm{MAX}_I) - 10 \times \log_{10}(\mathrm{MSE}) \tag{1}$$

where MAX_I is the maximum possible pixel value of the image, and MSE (mean squared error) is computed across all pixels. You may use skimage.metrics.peak_signal_noise_ratio for convenience. Report the average PSNR you get from the autoencoder across all images in the validation set (it should be around 15).

6   PyTorch (Extra Credit)

While you were able to derive manual backpropogation rules for sigmoid and fully-connected layers, wouldn't it be nice if someone did that for lots of useful primatives and made it fast and easy to use for general computation? Meet automatic differentiation. Since we have high-dimensional inputs (images) and low-dimensional outputs (a scalar loss), it turns out **forward mode AD** is very efficient. Popular autodiff packages include pytorch (Facebook), tensorflow (Google), autograd (Boston-area academics). Autograd provides its own replacement for numpy operators and is a drop-in replacement for numpy, except you can ask for gradients now. The other two are able to utilize GPUs to perform highly optimized and parallel computations, and are very popular for researchers who train large networks. Tensorflow asks you to build a computational graph using its API, and then is able to pass

data through that graph. PyTorch builds a dynamic graph and allows you to mix autograd functions with normal python code much more smoothly, so it is currently more popular in academia.

For **extra credit**, we will use PyTorch as a framework. Many computer vision projects use neural networks as a basic building block, so familiarity with one of these frameworks is a good skill to develop. Here, we basically replicate and slightly expand our handwritten character recognition networks, but do it in PyTorch instead of doing it ourselves. Feel free to use any tutorial you like, but we like the offical one or this tutorial (in a jupyter notebook) or these slides (starting from number 35).

For this section, you're free to implement these however you like. All of the tasks required here are fairly small and don't require a GPU if you use small networks. Include the neural network code for each part in the writeup.

6.1 Train a neural network in PyTorch

Q6.1.1 Code/Writeup [5 points] Re-write and re-train your fully-connected network on the included NIST36 in PyTorch. Plot training accuracy and loss over time.

Q6.1.2 Code/Writeup [5 points] Train a convolutional neural network with PyTorch on the included NIST36 dataset. Compare its performance with the previous fully-connected network.

Q6.1.3 Code/Writeup [5 points] Train a convolutional neural network with PyTorch on CIFAR-10 (`torchvision.datasets.CIFAR10`). Plot training accuracy and loss over time.

Q6.1.4 Code/Writeup [10 points] In Homework 1, we tried scene classification with the bag-of-words (BoW) approach on a subset of the SUN database. Use the same dataset in HW1, and implement a convolutional neural network with PyTorch for scene classification. Compare your result with the one you got in HW1, and briefly comment on it.

6.2 Fine Tuning

When training from scratch, a lot of epochs and data are often needed to learn anything meaningful. One way to avoid this is to instead initialize the weights more intelligently. These days, it is most common to initialize a network with weights from another deep network that was trained for a different purpose. This is because, whether we are doing image classification, segmentation, recognition etc..., most real images share common properties. Simply copying the weights from the other network to yours gives your network a head start, so your network does not need to learn these common weights from scratch all over again. This is also referred to as fine tuning.

Q6.2.1 Code/Writeup [5 points] Fine-tune a single layer classifier using pytorch on the flowers 17 (or flowers 102!) dataset using squeezenet1_1, as well as an architecture you've designed yourself (for example 3 convolutional layers followed 2 fully connected layers, it's standard slide 6) and trained from scratch. How do they compare?

We include a script in `scripts/` to fetch the flowers dataset and extract it in a way that torchvision.datasets.ImageFolder can consume it, see an example, from `data/oxford-flowers17`. You should look at how SqueezeNet is defined, and just replace the classifier layer. There exists a pretty good example for fine-tuning in PyTorch.

7   Deliverables

The assignment should be submitted to canvas. The writeup should be submitted as a pdf named <AndrewId>.pdf. The code should be submitted as a zip named <AndrewId>.zip. The zip when uncompressed should produce the following files.

- `nn.py`

- `q4.py`

- `run_q2.py`

- `run_q3.py`

- `run_q4.py`

- `run_q5.py`

- any `.py` files for Q6 (extra credit)

- `util.py`

References

[1] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. 2010. http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf.

[2] P. J. Grother. Nist special database 19 – handprinted forms and characters database. https://www.nist.gov/srd/nist-special-database-19, 1995.

8 Appendix: Neural Network Overview

Deep learning has quickly become one of the most applied machine learning techniques in computer vision. Convolutional neural networks have been applied to many different computer vision problems such as image classification, recognition, and segmentation with great success. In this assignment, you will first implement a fully connected feed forward neural network for hand written character classification. Then in the second part, you will implement a system to locate characters in an image, which you can then classify with your deep network. The end result will be a system that, given an image of hand written text, will output the text contained in the image.

8.1 Basic Use

Here we will give a brief overview of the math for a single hidden layer feed forward network. For a more detailed look at the math and derivation, please see the class slides.

A fully-connected network $\mathbf{f}$, for classification, applies a series of linear and non-linear functions to an input data vector $\mathbf{x}$ of size $N \times 1$ to produce an output vector $\mathbf{f}(\mathbf{x})$ of size $C \times 1$, where each element i of the output vector represents the probability of $\mathbf{x}$ belonging to the class i. Since the data samples are of dimensionality N, this means the input layer has N input units. To compute the value of the output units, we must first compute the values of all the hidden layers. The first hidden layer *pre-activation* $\mathbf{a}^{(1)}(\mathbf{x})$ is given by

$$\mathbf{a}^{(1)}(\mathbf{x}) = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$

Then the *post-activation* values of the first hidden layer $\mathbf{h}^{(1)}(\mathbf{x})$ are computed by applying a non-linear activation function $\mathbf{g}$ to the *pre-activation* values

$$\mathbf{h}^{(1)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(1)}(\mathbf{x})) = \mathbf{g}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$

Subsequent hidden layer $(1 < t \leq T)$ pre- and post activations are given by:

$$\mathbf{a}^{(t)}(\mathbf{x}) = \mathbf{W}^{(t)}\mathbf{h}^{(t-1)} + \mathbf{b}^{(t)}$$
$$\mathbf{h}^{(t)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(t)}(\mathbf{x}))$$

The output layer *pre-activations* $\mathbf{a}^{(T)}(\mathbf{x})$ are computed in a similar way

$$\mathbf{a}^{(T)}(\mathbf{x}) = \mathbf{W}^{(T)}\mathbf{h}^{(T-1)}(\mathbf{x}) + \mathbf{b}^{(T)}$$

and finally the *post-activation* values of the output layer are computed with

$$\mathbf{f}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(T)}(\mathbf{x})) = \mathbf{o}(\mathbf{W}^{(T)}\mathbf{h}^{(T-1)}(\mathbf{x}) + \mathbf{b}^{(T)})$$

where $\mathbf{o}$ is the output activation function. Please note the difference between $\mathbf{g}$ and $\mathbf{o}$! For this assignment, we will be using the sigmoid activation function for the hidden layer, so:

$$\mathbf{g}(y) = \frac{1}{1 + \exp(-y)}$$

Figure 2: Samples from NIST Special 19 dataset [2]

where when $\mathbf{g}$ is applied to a vector, it is applied element wise across the vector.

Since we are using this deep network for classification, a common output activation function to use is the softmax function. This will allow us to turn the real value, possibly negative values of $\mathbf{a}^{(T)}(\mathbf{x})$ into a set of probabilities (vector of positive numbers that sum to 1). Letting $\mathbf{x}_i$ denote the ith element of the vector $\mathbf{x}$, the softmax function is defined as:

$$\mathbf{o}_i(\mathbf{y}) = \frac{\exp(\mathbf{y}_i)}{\sum_j \exp(\mathbf{y}_j)}$$

Gradient descent is an iterative optimisation algorithm, used to find the local optima. To find the local minima, we start at a point on the function and move in the direction of negative gradient (steepest descent) till some stopping criteria is met.

8.2 Backprop

The update equation for a general weight $W_{ij}^{(t)}$ and bias $b_i^{(t)}$ is

$$W_{ij}^{(t)} = W_{ij}^{(t)} - \alpha * \frac{\partial L_{\mathbf{f}}}{\partial W_{ij}^{(t)}}(\mathbf{x}) \qquad b_i^{(t)} = b_i^{(t)} - \alpha * \frac{\partial L_{\mathbf{f}}}{\partial b_i^{(t)}}(\mathbf{x})$$

α is the learning rate. Please refer to the backpropagation slides for more details on how to derive the gradients. Note that here we are using softmax loss (which is different from the least square loss in the slides).