Deep Learning (IT-3030)
Project #1: Implementing Backpropagation

This document describes the first assignment in IT3030, where you will implement backpropagation for deep neural networks. By doing this project, you will: a) gain a deep understanding of backpropagation via hands-on implementation of many of its more complex details, and b) become familiar with advanced matrix operations in numpy (or similar packages in Python or another language).

The rules are as follows:

- The task must be solved individually.
- It must be solved in Python (or a similar language) **without** using software that is dedicated to deep learning, such as Tensorflow, Keras or PyTorch.
- Your solution will be given between 0 and 10 points; the rules for scoring are listed in the last section (Deliverables).
- Demonstrations for this project will take place during the week of February 12 -16, 2024. Information regarding the exact times for demos, possible signups, etc. will be posted on Blackboard.
- Code must be properly commented and uploaded to Blackboard prior to or (immediately) after your demonstration session.

1 Introduction

The heart and soul of deep learning is a nearly-40-year-old algorithm known as **backpropagation**. Though numerous *bells and whistles* have supplemented backpropagation over the years, with some of these enhancements proving crucial to the scaling up of neural networks to truly **deep** learning, the core learning algorithm and framework is still backprop.

Although packages such as Tensorflow and PyTorch take (almost) all of the drudgery out of deep learning by abstracting away the essence of backpropagation – the calculation of gradients linking weights to loss – an understanding of these details seems quite essential to anyone seriously considering a career in AI and Machine Learning. That understanding stems most readily from low-level implementation, and the skills thereby achieved provide a solid base for deeper investigations and actual AI research. It is not unusual for a tech employer to query your knowledge of such details during an interview for an AI position.

Your system will enable users to build and run networks that have a wide range of sizes and types, from simple input-output networks with no hidden layers to those with up to 5 hidden layers. A limited variety of activation functions and regularization schemes will also be at the user's disposal.

As described in later sections, your system will provide a module for generating datasets, and you will define a format and parser for network configuration files. These, along with a few simple visualization tools (for the data images and learning progress), constitute peripheral support for the two core processes of the backpropagation algorithm: the forward and backward passes.

2 The Neural Network

The networks built and run in your system will follow the basic topology of Figure 1, with an input layer, anywhere from 0 to 5 hidden layers, and a final softmax output layer. Your system must provide the softmax layer as an option, since the tasks of this assignment involve classification. However, your system must also accept standard (non-softmax) output layers in support of regression tasks.

Note that weights (and biases) are typically associated with the layer that they feed **into**, so a straightforward object-oriented implementation will bundle the units (a.k.a. neurons) of a layer with the incoming weights and biases, all into a layer object.

You must implement your system in an object-oriented manner. At the very least, your system must include a **network** and a **layer** object, and each must have a **forward_pass** and **backward_pass** method. You must also implement your data generator as an object that returns data sets when given image-defining parameters such as size (n), noise level, etc.

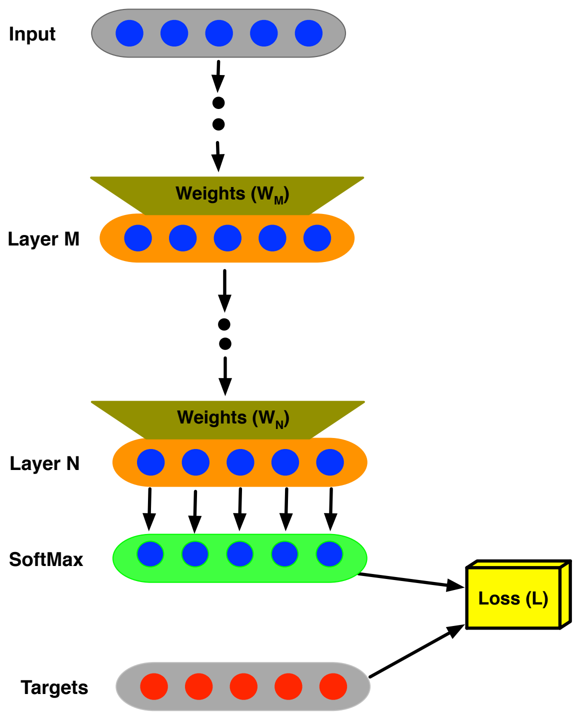

Figure 1: A typical topology for a classifier network built by your system.

3 The Forward Pass

The forward pass involves the following steps:

1. Fetch a minibatch of training cases.
2. Send each case through the network, from the input to the output layer. At each layer (L), multiply the outputs of the upstream layer by the weights and then add in the biases. Finally, apply the activation function to these sums to produce the outputs of L.
3. Apply the softmax function to the values entering the output layer to produce the network's outputs. Remember that softmax has no incoming weights.
4. Compare the targets to the output values via the loss function.
5. Cache any information (such as the outputs of each layer) needed for the backward stage.

You are free to send cases through the network individually or in minibatch chunks. The latter requires a bit more attention to the details of matrix operations, but the added flexibility (and speed) greatly improves your system. There is no point loss or gain for either approach. For those interested in adapting matrix operations to fit a wide range of dimensions, check out numpy's **einsum** function.

Aside from the potential complexity of the matrix operations, the forward pass should be a simple process that requires very little code (and runtime).

4 The Backward Pass

As with the forward pass, the backward pass can be performed one case at a time or with the entire minibatch in one shot. Regardless, the weights and biases should not be updated until every case in the minibatch has been passed back through the net and its gradients calculated.

Most important details of the backpropagation algorithm appear in the lecture notes and will not be repeated here. However, a few points are not covered in those notes and/or are worth repeating or summarizing.

For each activation function supported by your system (e.g. sigmoid, tanh, relu, linear), a corresponding derivative function must be written. Similarly, a derivative function must exist for each of your loss functions (e.g. mean-squared-error and cross-entropy) and for the softmax operator.

The backward pass involves the following steps:

1. Compute the initial Jacobian (J_S^L) representing the derivative of the loss with respect to the network's (typically softmaxed) outputs.
2. Pass J_S^L back through the Softmax layer, modifying it to J_N^L, which represents the derivative of the loss with respect to the outputs of the layer prior to the softmax, layer N.
3. Pass J_N^L to layer N, which uses it to compute its delta Jacobian, δ_N.
4. Use δ_N to compute: a) weight gradients J_W^L for the incoming weights to N, b) bias gradients J_B^L for the biases at layer N, and c) J_{N-1}^L to be passed back to layer N-1.

5. Repeat steps 3 and 4 for each layer from N-1 to 1. Nothing needs to be passed back to the Layer 0, the input layer.
6. After all cases of the minibatch have been passed backwards, and all weight and bias gradients have been computed and accumulated, modify the weights and biases.

Most of the backward pass activities can be achieved with numpy's tensor operations, such as **dot, outer,** and **einsum**, although you will need to pay special attention to the tensors, their transposes, and vectors to insure proper results.

5 Regularization

Your system will include an option for using a regularizer for weights and biases. You must provide two possibilities for the regularizing function: L1 or L2, and a single regularization constant should pertain to the entire network. As described in the lecture notes, $\frac{\partial\omega(\theta)}{\partial w}$ (the derivative of the regularizing sum with respect to any weight) is different for L1 and L2. For L1, $\frac{\partial\omega(\theta)}{\partial w} = c \times sign(w)$, but for L2, $\frac{\partial\omega(\theta)}{\partial w} = c \times w$. In both cases, c is the constant regularization rate, which is typically a small fraction such as 0.001.

6 Data Generation

You must build (or download) a system for generating two-dimensional, binary, pixel images of your choosing. The mandatory criteria are the following:

1. Image dimensions are n x n, for all n where $10 \leq n \leq 50$; the user can choose n when requesting a new image set.
2. At least 4 different classes of images can be generated, e.g. circles, rectangles, triangles, and crosses; or a's, b's, q's and x's, or any other type of images that you prefer.
3. Any generated image set should have approximately the same number of images of each class.
4. A user-specified noise parameter will control the fraction of randomly-set pixels in each image.
5. The generator will normally return 3 image sets corresponding to training, validation and testing, with the relative sizes of each also specified by the user – for example 70% training, 20% validation, 10% testing.
6. A user-specified size range for the height and width of the image (compared to the n x n background) will support images of the same class but of different sizes.
7. A flattening option is included such that images can be returned either as 2-d arrays or as vectors.

Figure 2 provides examples of the types of images that your generator might produce. Crucially, the module must be able to produce a large number of **unique** images of the same type but within the same n x n frame. Thus, an image such as a rectangle should be possible to draw at different locations, in different sizes, and with different ratios of height to width. Similarly, a cross might vary in size or thickness. In addition, the noise parameter should further support uniqueness even when the base (noiseless) image repeats many times. There is no strict requirement for the manner in which your system produces unique (or nearly unique) images, and the occasional repeat of an image is not strictly prohibited, but diversity is mandatory.

One key reason that MNIST images are relatively easy to classify is that all digits are **centered** on the background. Hence, you should insure that some combination of user-specified parameters will enforce centering, when desired; but you must also permit the production of uncentered images in order to test the generality of your deep network.

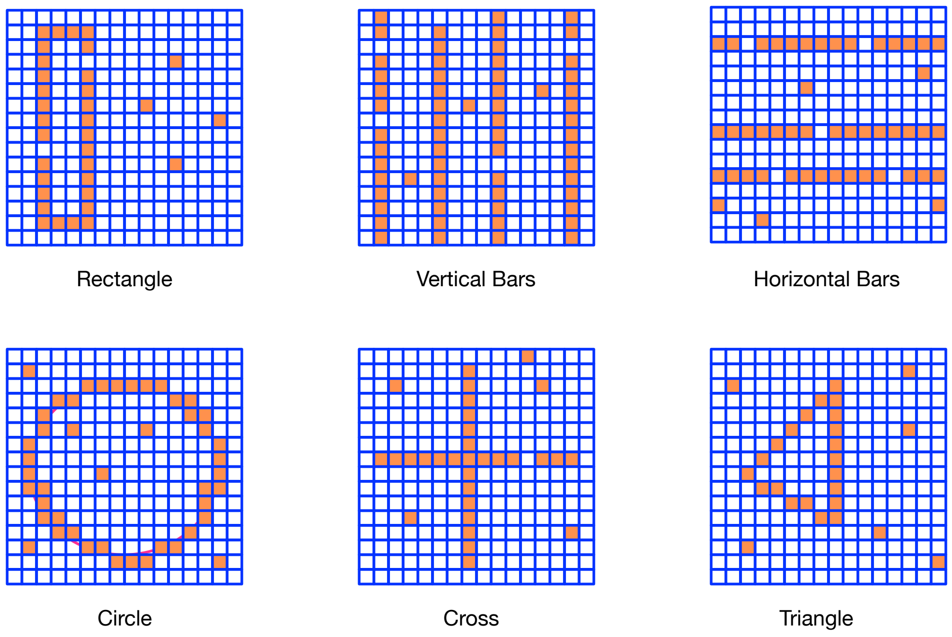

Figure 2: Examples of 16 x 16 images of 6 different patterns. Note the appearance of noise in the images.

6.1 The Doodler

We provide the file *doodler_forall.py*, which contains a **Doodler** class that can generate images of 9 different object types: ball, ring, frame, box, flower, bar, polygon, triangle and spiral. Some experimentation is necessary to find a subset of the 9 object types that gives good results for your system, and there are many combinations of arguments to the function **gen_standard_cases** that you will surely want to try.

The Doodler fulfills most of the above requirements for the data generator, although it only returns a list of images, their target vectors and their labels; they are not separated into training, validation and test sets. Doodler case sets are represented as 5-item tuples, which can be saved to and loaded from a file, via functions **dump_doodle_cases** and **load_doodle_cases**, respectively. See *doodler_forall.py* for more details.

In general, the Doodler handles most of the data-generation work required for this project, but there are surely many other options available online. And some students will probably prefer to write their own system from scratch, since they then will have full control of image production.

7 Network Architecture Configuration Files

You will design a format for files that specify the complete architecture of a deep network, including the number and size of each layer, the activation function used by each layer, the range of initial weights, the loss function, regularizers, learning rates, etc. Each network run within your system will be specified in one of these architecture/configuration files, and thus, your system must be capable of parsing a wide variety of such files and then producing and running the networks that they specify.

During the demonstration session, you will be asked to provide one or several of these files, which one of the instructors or assistants will then modify and attempt to run through your system. Typical modifications include the addition or removal of several layers, changes to activation functions or learning rates, and modifications of initial-weight ranges. Modifications will be made with your own assistance in order to uphold (your) proper syntax, but flexibility with respect to all of the **pivotal** parameters specified below must be displayed by your system.

These files must consist solely of simple data (to be interpreted by your system), not a set of commands (such as *build_layer(...)*). They should be very easy for humans to read, modify and parse. The following is a simple example of a style that uses keywords GLOBALS and LAYERS to separate the specification of parameters that pertain to the whole network from those specific to a given layer. In this format, all keywords end with a colon.

```
GLOBALS
loss: cross_entropy lrate: 0.1 wreg: 0.001 wrt: L2
LAYERS
input: 20
size: 100 act: relu wr: (-0.1 0.1) lrate: 0.01
size: 5 act: relu wr: glorot br: (0 1)
type: softmax
```

The network specified above consists of 4 layers. The first, input, contains 20 neurons. The two hidden layers (of sizes 100 and 5) both use RELU for activation. The first uses an initial range for its incoming weights of (-0.1 0.1), while the second relies on the glorot initialization algorithm. Also, the first hidden layer specifies a private learning rate (for its incoming weights), while the second hidden layer does not (and will therefore use the global learning rate of 0.1. However, it does specify a range of (0,1) for its biases. The final layer performs softmax and thus has no incoming weights nor learning rate; its size is assumed to be that of its upstream neighbor, i.e. 5.

The global parameters listed above are the loss function, the global (i.e. default) learning rate, the weight regularization rate (wreg) and the weight regularizing type (wrt).

The system that parses this type of file can handle missing data via a simple set of default values and procedures. For example, the default layer type is *basic_layer*, and even input layers use this type. Only the softmax layer requires an explicit *type:* keyword.

Note, the above file format and conventions are just an example, and one that is not entirely trivial to implement. You are free to design your own, with or without keywords, defaults, etc. You are also free to use modules such as **configparser** in Python or similar modules in other languages as a component of your own parser.

7.1 Pivotal Parameters

Each of the following parameters must be specifiable in your architecture configuration files and supported by your deep learning system.

1. The number of layers. This may be indirectly specified by the listing of the layers themselves. Your system must handle anywhere from 0 to 5 hidden layers.
2. The number of neurons in each layer, including the input and output layers. Sizes from 1 to 1000 must be possible.
3. The activation function used in each layer (except the input layer). The options must include: sigmoid, tanh, relu, and linear.
4. Softmax must be a possible option for the output layer. It need not be applicable anywhere else in the network.
5. The loss function. Options must include mean-squared error (MSE) and cross-entropy, but feel free to add more.
6. Global weight regularization options: L1, L2, none, along with the global weight regularization rate (typically a small constant fraction such as 0.001).
7. Initial weight ranges for each (non-input) layer. You may (or may not) support use of the glorot initializer; it is a very effective approach.
8. The dataset to be run. You may choose to pre-generate datasets and simply include the file name in the configuration file, or you may include all of the parameters for the data generator in the configuration file. You only need to provide one of these options to the user, but you must show the generation of data cases during the demonstration session.

8 Visualization

You must build (or download) an image viewer that allows the user to display up to 10 random 2-D images from any image set. Matplotlib and other packages provide a host of tools for displaying arrays; use them freely. In addition, the Doodler includes code for displaying 2d images. In calls to **gen_standard_cases**, set the keyword argument *show* to True, and you will see each of the images in its own window. The main display function is **quickplot_matrix**, which you can easily co-opt for other purposes. You can also call **show_doodle_cases** to display the images and classifications of any collection of cases. See comments in the *doodler_forall.py* for more details.

In addition, your system must generate a plot of learning progress that shows the number of epochs (or minibatches) on the X axis, and the loss (error) on the Y axis. The plot should contain one curve for the training data and one for the validation data. To display the final testing error, either include it on the plot or print it on the command line. Figure 3 shows a typical example produced with a combination of matplotlib primitives.

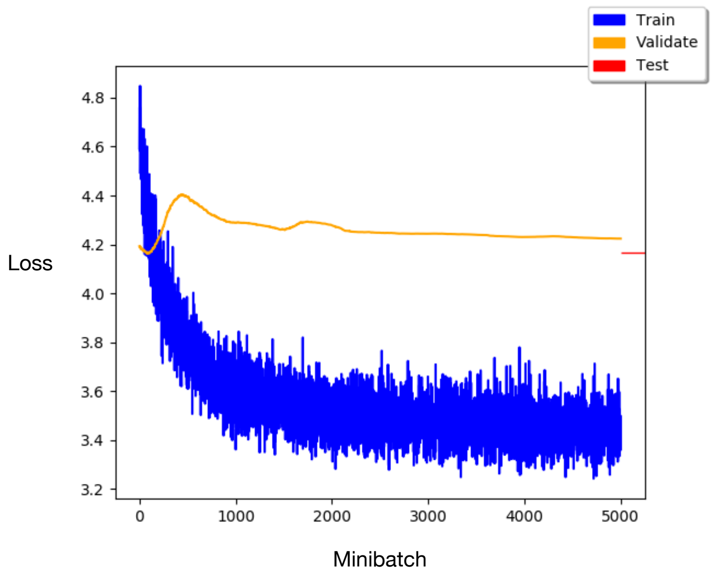

Figure 3: Standard plot of loss progression for the learning, validation, and test sets.

9 Deliverables

The subsystems and procedures listed below will be demonstrated by running the system multiple times and by discussing your code with an instructor or assistant. You must come to the demo session with a few fully-functioning configuration files; the reviewer will then modify these files in various ways and run them through your system. You will assist the reviewer in these modifications in accordance with the legal syntax of your configuration files and any other constraints of your system. However, these constraints should not exclude any of the requirements listed in this document, many of which the reviewer will explore in your system.

If your data-generating system takes several minutes to produce a few hundred data cases, then you should produce and save to file at least one large (500 or more cases) dataset for use during the demo session. Regardless, you will be asked to demonstrate your data generator during the demonstration session, even if that only involves generating (and displaying) 10 or 20 cases.

The point breakdown is as follows:

1. The data generator, image viewer, and graphic display of learning progress. (**0.5 points**).
2. The (verbal) description of the format for configuration files, along with code to parse those files. (**0.5 points**).
3. The forward pass of the deep network, with core functions performed with matrix and vector operations in numpy (or a similar package). This entails analysis of your source code and running a few different networks (chosen by the evaluator) for a few minibatches and printing out all of the following: network inputs, network outputs, target values, and error/loss. Be sure that your system has a **verbose** flag that, when True, allows all of this information to be printed out to the command line. (**3 points**).
4. The backward pass (backpropagation) for modifying the network weights and biases. This must clearly perform the explicit computation of the key Jacobian matrices and vectors, as specified in the lecture

notes. This entails analysis of your source code along with running several different networks on several different data sets (all chosen by the evaluator) for hundreds or thousands of minibatches and then analyzing the plots of learning progress (as in Figure 3) and verifying that, indeed, the system is learning the training data. The actual performance (i.e. error/loss) on the validation and test data sets will not affect your point score, as long as the system runs properly (i.e. no crashes or obvious mistakes) on the validation and test sets, and displays their results. (**6 points**).

9.1 The Demonstration Session

To insure a smooth demonstration, you should have the following readily available during your session:

1. A configuration file for a network with at least 2 hidden layers that runs on a dataset of at least 500 training cases and that has previously shown some learning progress (i.e. loss clearly declines over time / minibatches). All parameters of this network should be tuned to those that have worked well in the past.
2. A configuration file for a network with no hidden layers that runs on the same 500-item training set as above. This network may or may not exhibit learning progress.
3. A configuration file for a network with at least 5 hidden layers that runs on a dataset of at least 100 training cases for a minimum of 10 passes through the entire training set.

Demonstration sessions typically last 20-30 minutes.

10 Important Practical Details

The code for networks, layers and the data-set generator must be object-oriented. Failure to satisfy this simple criteria will result in considerable point loss.

WARNING: Failure to properly explain ANY portion of your code (or to convince the reviewer that you wrote the code) can result in the loss of 3 to 10 points, depending upon the seriousness of the situation. This is an individual exercise in programming, not in downloading nor copying.

A zip file containing your commented code must be uploaded to BLACKBOARD before or (immediately) after your demonstration. You will not get explicit credit for the code, but it is crucial that we have the code online in the event that you decide to register a formal complaint about your grade (for the entire course).

The 10 points for this project are 10 of the 30 project points that are available for the entire semester. To pass this particular project, you must receive at least 3 points.