COMP0088
Introduction to Machine Learning
Lab Assignment 5

Matthew Caldwell

October 29, 2021

Task Summary

Week 5 Notes

In this week's assignment you are asked to implement a complete **multi-layer perceptron** model, including activation and loss functions, a forward pass to process data, a backwards pass to calculate gradients, mini-batch gradient descent optimisation of the weights and finally

prediction from new data. There are a lot of pieces here, but don't be intimidated: *most* of them are pretty straightforward and we'll take it step by step.

For simplicity and transparency, this implementation will use just basic data structures (Python lists and dictionaries, NumPy arrays). In practice this would usually be built into a more organised class structure—we will see examples of this in week 6. If you feel particularly enthusiastic you are welcome to re-engineer these functions into something more elegant.[1]

The assignment script is `week_5.py`, generating output file `week_5.pdf`. There are some potentially-useful command line options, which you can list using:

```
$ python week_5.py -h
```

By default the script uses the same synthetic data as the week 3 exercises, loaded from the file `week_3_data.csv`. Example output from a completed assignment is shown in Fig. 1. Plotting functionality is provided, so your output should look similar, though by default the resolution for the class maps will be significantly lower (to avoid scaling issues).

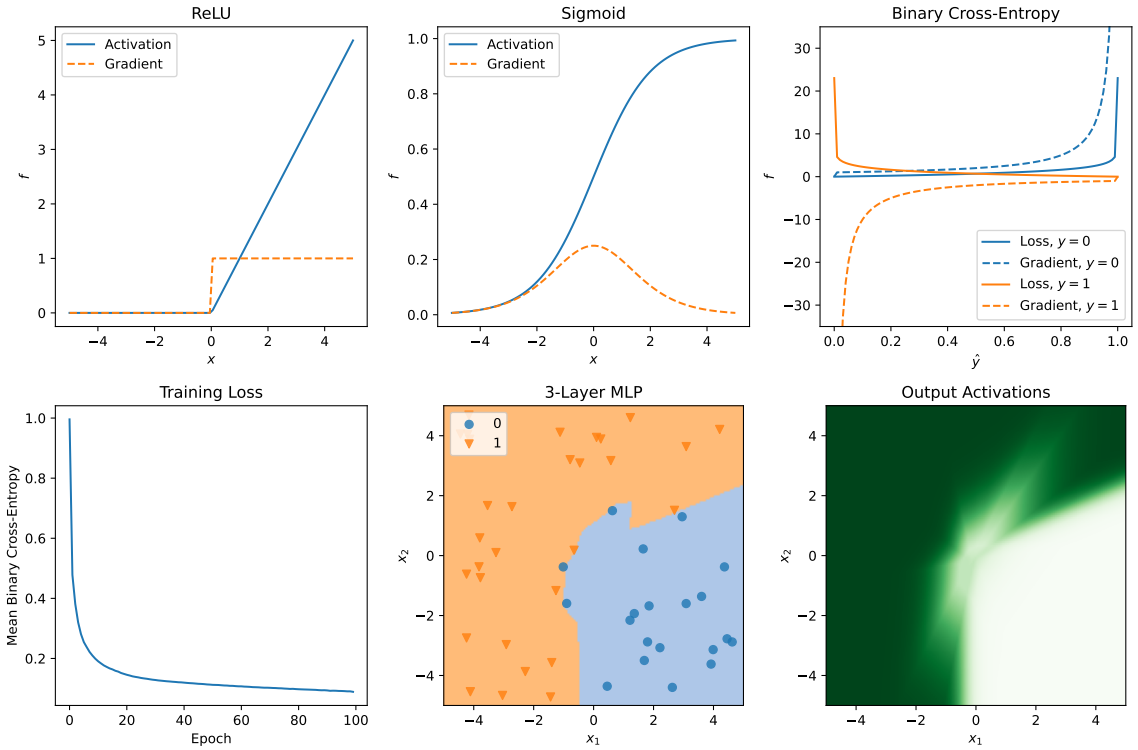

Figure 1: Example output from the `week_5.py` script

[1] There are no other "further exploration" tasks this week—if you run out of things to do here, reward yourself with some rest and relaxation!

1 Activation functions

The MLP for these exercises will be a simple binary classifier with two input features and a single scalar (probability) output. It will use **ReLU** activations for the intermediate hidden layers of the network and a combination of a **sigmoid** activation on the final layer with **binary cross-entropy** for calculating the loss. You will therefore need forward and derivative implementations of all three functions.

ReLU is defined as:

$$\text{relu}(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

This is not differentiable at zero, but we can define a subgradient:

$$\text{relu}'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

The sigmoid function is:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3}$$

with derivative:

$$\sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big) \tag{4}$$

The binary cross-entropy loss is:

$$L(y, \hat{y}) = -\big[y \log \hat{y} + (1 - y) \log(1 - \hat{y})\big] \tag{5}$$

with derivative:

$$L'(y, \hat{y}) = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})} \tag{6}$$

Note that for the purposes of this implementation, all of these functions—including the loss—are to be calculated **elementwise** and should return an array of the same size and shape as the input. (For the cross-entropy functions, which have two inputs, use the shape of the `y_hat` argument.)

1.1 Implement forward and derivative ReLU activation

Provide bodies for the following pair of functions in `week_5.py`:

```
def relu ( z ):
    """
    Rectified linear unit activation function.

    # Arguments
        z: a single number or numpy array

    # Returns
        r: a number or numpy array of the same dimensions
            as the input value, giving the ReLU of
            of each input value
    """
    # TODO: implement this
    return None
```

```
def d_relu ( z ):
    """
    Gradient of the ReLU function

    # Arguments
        z: a single number or numpy array

    # Returns
        r: a number or numpy array of the same dimensions
            as the input value, giving the gradient
            of the ReLU function at each input value
    """
    # TODO: implement this
    return None
```

1.2 Implement forward and derivative sigmoid activation

Provide bodies for the following pair of functions in `week_5.py`:

```
def sigmoid ( z ):
    """
    Sigmoid activation function.

    # Arguments
        z: a single number or numpy array

    # Returns
        r: a number or numpy array of the same dimensions
            as the input value, giving the sigmoid (logistic)
            output for each input value
    """
    # TODO: implement this
    return None

def d_sigmoid ( z ):
    """
    Gradient of the sigmoid function

    # Arguments
        z: a single number or numpy array

    # Returns
        r: a number or numpy array of the same dimensions
            as the input value, giving the gradient
            of the sigmoid function at each input value
    """
    # TODO: implement this
    return None
```

1.3 Implement forward and derivative binary cross-entropy

Provide bodies for the following pair of functions in `week_5.py`:

```
def binary_crossentropy_loss ( y, y_hat, eps=1e-10 ):
    """
    Binary cross-entropy loss for predictions, given the
    true values.

    # Arguments:
        y: a numpy array of true binary labels.
        y_hat: a numpy array of predicted labels,
            as numbers in open interval (0, 1). must have
            the same number of entries as y, but not
            necessarily identical shape
        eps: a small offset to avoid numerical problems
            when predictions are very close to 0 or 1

    # Returns:
        loss: a numpy array of individual cross-entropy
            loss values for each prediction. will be
            the same shape as y_hat irrespective of the
            shape of y
    """
    # TODO: implement this
    return None

def d_binary_crossentropy_loss ( y, y_hat, eps=1e-10 ):
    """
    Gradient of the cross-entropy loss for predictions, given the
    true values.

    # Arguments:
        y: a numpy array of true binary labels.
        y_hat: a numpy array of predicted labels,
            as numbers in open interval (0, 1). must have
            the same number of entries as y, but not
            necessarily identical shape
        eps: a small offset to avoid numerical problems
            when predictions are very close to 0 or 1

    # Returns:
        grad: a numpy array of individual cross-entropy
            gradient values for each prediction. will be
            the same shape as y_hat irrespective of the
            shape of y
    """
    # TODO: implement this
    return None
```

NB: these functions take two arguments, but the shape of `y_hat` takes precedence. The `y` argument may not be the same shape, so you should `reshape` it appropriately first.

2 Build a neural network

The underlying data structures for the mutli-layer perceptron are **layers**, represented by a Python dictionary, and the whole network, which is just a Python list of layers. Each layer performs of a weighted sum of its inputs followed by a non-linear activation.

2.1 Create and initialise a single network layer

Implement the following function in `week_5.py`:

```
def init_layer ( fan_in, fan_out, act, rng ):
    """
    Create a single neural network layer.

    # Arguments
        fan_in: the number of incoming connections
        fan_out: the number of outgoing connections
        act: name of the activation function for the
            layer, either "sigmoid" or "relu"
        rng: an instance of numpy.random.Generator
            from which to draw random numbers

    # Returns
        layer: a dict holding the layer contents, with
            keys 'W', 'b', 'shape' and 'act'.
            (See the coursework for full details.)
    """
    # TODO: implement this
    return None
```

The values assigned to the four dictionary keys should be as follows:

- `W`: a weights matrix of shape `fan_in` $\times$ `fan_out`. We are going to be using ReLU activation for the hidden layers, so you should use He initialisation for the weight values, drawing from a uniform distribution over the range $(-\sqrt{6/\text{fan\_in}}, \sqrt{6/\text{fan\_in}})$.
- `b`: a bias vector of length `fan_out`. Biases can be initialised to 0.
- `act`: the name of the specified activation function, i.e. this is the same as the argument `act`.
- `shape`: a text string describing the shape mapping of the layer. This will be used to print information about a created MLP; you can format it anyway you like.

Note that the dictionary keys (and all the others that will be added later) are just text strings.

2.2 Create and initialise a complete network

Implement the following function in `week_5.py`:

```
def init_mlp ( spec, rng ):
    """
    Build a neural network according to the
    given specification.
```

```
    # Arguments
        spec: an iterable of tuples (fan_in, act)
            specifying the configuration of the network layers.
            there must be at least 2 elements; the last is only
            used to determine output size of the layer before,
            it does not create a layer of its own
        rng: an instance of numpy.random.Generator
            from which to draw random numbers

    # Returns
        mlp: a list of layer dicts
    """
    assert(len(spec) > 1)

    # TODO: implement this
    return None
```

An MLP network is just a list of the layers returned by `init_layer`. Iterate over the list of tuples in `spec` and create a layer for each one except the last, setting the `fan_out` of each layer as the `fan_in` of the next. (Use the activation from the input tuple; the last one will have an activation of `None`.)

3 Perform a forward pass

In the forward pass through the network, layers receive input data and produce outputs, storing information they will need to calculate gradients. The outputs from each layer become the inputs to the next.

3.1 Perform a forward pass through a single network layer

Implement the following function in `week_5.py`:

```
def layer_forward ( layer, X ):
    """
    Run a forward pass of data through a layer, storing
    intermediate values in the layer dict.

    # Arguments
        layer: a layer dict as created by init_layer
        X: the input data to the layer, a matrix
            where the columns are features and the
            rows are samples. feature count must
            match the layer's fan_in

    # Returns
        A: the layer's output activations, a matrix where the
            columns are (fan_out) features and the rows are
            samples
    """
    assert(x.shape[-1] == layer['W'].shape[0])

    # TODO: implement this
    return None
```

The forward pass for a single layer computes

$$\mathbf{Z} = \mathbf{X}\mathbf{W} \tag{7}$$

$$\mathbf{A} = \text{activation}(\mathbf{Z}) \tag{8}$$

You will need the input data, linear combination and output for the backwards pass, so store these in the `layer` dictionary with the keys `X`, `Z` and `A`, respectively.

3.2 Perform a forward pass through a whole network

Implement the following function in `week_5.py`:

```
def mlp_forward ( mlp, X ):
    """
    Run a forward pass through a whole neural net.

    # Arguments
        mlp: a list of layer dicts, as created by init_mlp
        X: the input data to the network, a matrix
            where the columns are features and the
```

```
            rows are samples. feature count must
            match the first layer's fan_in

    # Returns
        A: the output activations of the final network layer
    """
    # TODO: implement this
    return None
```

The forward pass through the whole network just iterates over the layers in turn, calling `layer_forward` for each one. The input to the first layer is the network input data argument `X`, while the input for subsequent layers is the output of the layer before.

4 Perform a backward pass

In the backward pass through the network, layers receive the downstream loss gradient (with respect to their output activations) and calculate their loss gradients with respect to their weights, bias and inputs, storing these for subsequent learning. The gradients with respect to the inputs from each layer are then propagated back as the downstream gradients for the previous layer.

4.1 Perform a backward pass through a single network layer

Implement the following function in `week_5.py`:

```
def layer_backward ( layer, dA ):
    """
    Run a backward pass of gradients through a layer, storing
    computed values in the layer dict. The forward pass must
    have been performed first.

    # Arguments
        layer: a layer dict as created by init_layer
        dA: the gradients of the loss with respect to the
            forward pass activations. a matrix the same shape
            as those previously computed activations.

    # Returns
        dX: the gradients of the loss with respect to the
            layer inputs from the forward pass
    """
    assert(dA.shape == layer['A'].shape)

    # TODO: implement this
    return None
```

The gradients propagate backwards by multiplication. For the activations, the derivative evaluation and multiplication both occur **elementwise**:

$$\nabla_{\mathbf{Z}} = \nabla_{\mathbf{A}} \times \text{activation}'(\mathbf{Z}) \tag{9}$$

For the weights and inputs, the calculation involves inner products, so we can express it using matrix multiplication, taking care that the right product winds up in the right place in the matrix. The relevant expressions are:

$$\nabla_{\mathbf{X}} = \nabla_{\mathbf{Z}} \mathbf{W}^{\mathsf{T}} \tag{10}$$

$$\nabla_{\mathbf{W}} = \mathbf{X}^{\mathsf{T}} \nabla_{\mathbf{Z}} \tag{11}$$

For the bias term, there is no input dependence, and so the gradient vector is $\mathbf{1} \cdot \nabla_{\mathbf{Z}}$ (i.e, the columnwise sum of $\nabla_{\mathbf{Z}}$).

Save all the gradients in the dictionary with keys `dA`, `dZ`, `dW`, `db` and `dX`, returning the latter.

NB: while this function can be implemented in just a few lines, there's a lot going on here, so it's worth taking some time to understand it. This is the core of the backpropagation process that makes it possible to train a neural networks.

4.2 Perform a backward pass through a whole network

Implement the following function in `week_5.py`:

```
def mlp_backward ( mlp, d_loss ):
    """
    Backpropagate gradients through the whole neural net.
    The forward pass must have been performed first.

    # Arguments
        mlp: a list of layer dicts, as created by init_mlp
        d_loss: the gradients of the loss at the final
            layer output, a matrix the same shape
            as previously computed activations.

    # Returns
        None
    """
    # TODO: implement this
    pass
```

Like the forward pass, the backward pass simply involves iterating through the network calling layer_backward on each layer. Remember to traverse the list in reverse, and pass the gradient returned by each layer to the one before it.

5 Update the network weights

We'll train the network using vanilla gradient descent, simply adjusting the weights and bias at each iteration by a fraction of the gradient:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla_{\mathbf{W}} \tag{12}$$

$$\mathbf{b} \leftarrow \mathbf{b} - \alpha \nabla_{\mathbf{b}} \tag{13}$$

where α is the learning rate.

5.1 Update the weights for a single network layer

Implement the following function in `week_5.py`:

```
def layer_update ( layer, lr ):
    """
    Update layer weights & biases according to the previously
    computed gradients. Forward and backward passes
    must both have been performed.

    # Arguments
        layer: a layer dict as created by init_layer
        lr: the learning rate

    # Returns
        None
    """
    # TODO: implement this
    pass
```

Remember to update the values assigned to the keys `W` and `b` in the layer dictionary.

5.2 Update the weights for a whole network

Implement the following function in `week_5.py`:

```
def mlp_update ( mlp, lr ):
    """
    Update all network weights & biases according to the
    previously computed gradients. Forward and backward passes
    must both have been performed.

    # Arguments
        mlp: a list of layer dicts, as created by init_mlp
        lr: the learning rate

    # Returns
        None
    """
    # TODO: implement this
    pass
```

Again, this is just an iteration over the list (in any order) calling `layer_update`.

6 Train the network

We're going to train the network using **mini-batch gradient descent**. This means iterating over the training data in bite size chunks, and for each chunk executing a forward pass, then a backward pass, then a weights update. One complete run through all the mini-batches in the training set is known as an **epoch**.

6.1 Train the network on a single mini-batch of data

Implement the following function in `week_5.py`:

```
def mlp_minibatch ( mlp, X, y, lr ):
    """
    Fit a neural network to a single mini-batch
    of training data.

    # Arguments
        mlp: a list of layer dicts, as created by init_mlp
        X: an array of sample data, where rows are samples
            and columns are features. feature dimension must
            match the input dimension of mlp.
        y: vector of binary class labels corresponding to the
            samples, must be same length as number of rows in X
        lr: the learning rate

    # Returns
        loss: the mean training loss over the minibatch
    """
    assert(X.shape[0] == len(y))
    assert(X.shape[-1] == mlp[0]['W'].shape[0])

    # TODO: implement this
    return None
```

The data in `X` and `y` is for one single chunk, so you can just pass it straight to the network.

The output of `mlp_forward` is from the final layer sigmoid, representing the probabilistic predictions $\hat{\mathbf{y}}$. You will need to use this together with the ground truth labels `y` to calculate the cross entropy loss and also the loss gradient. The latter should then be propagated back through the network, and the former should be returned. (Divide by the number of samples to get the mean loss.)

6.2 Train the network for one complete epoch

Implement the following function in `week_5.py`:

```
    """
    Fit a neural network for one epoch -- ie, a single
    pass through the data in minibatches of specified size.

    # Arguments
        mlp: a list of layer dicts, as created by init_mlp
```

```
        X: an array of sample data, where rows are samples
            and columns are features. feature dimension must
            match the input dimension of mlp.
        y: vector of binary class labels corresponding to the
            samples, must be same length as number of rows in X
        batch: the size of minibatches to train
        lr: the learning rate
        rng: an instance of numpy.random.Generator
            from which to draw random numbers

    # Returns
        loss: the mean training loss over the whole dataset
    """
    # TODO: implement this
    return None
```

This is basically an exercise in data management, splitting the data and labels into matched chunks to pass to `mlp_minibatch`. You may find the `permutation` function useful to randomise the order of samples each epoch.

6.3 Train the network for multiple epochs

Implement the following function in `week_5.py`:

```
    """
    Fit a neural network iteratively for multiple
    epochs.

    # Arguments
        mlp: a list of layer dicts, as created by init_mlp
        X: an array of sample data, where rows are samples
            and columns are features. feature dimension must
            match the input dimension of mlp.
        y: vector of binary class labels corresponding to the
            samples, must be same length as number of rows in X
        batch: the size of minibatches to train
        epochs: number of epochs to train
        lr: the learning rate
        rng: an instance of numpy.random.Generator
            from which to draw random numbers

    # Returns
        loss: a list of the mean training loss at each epoch
    """
    # TODO: implement this
    return None
```

This should just call `mlp_epoch` the requested number of times, recording the loss history as it goes.

7 Test the network

The `mlp_forward` function already allows you to run test data through a network, but it is useful also to be able to generate class labels rather than probabilities.

7.1 Make predictions

Implement the following function in `week_5.py`:

```
def mlp_predict ( mlp, X, thresh=0.5 ):
    """
    Make class predictions from a neural network.

    # Arguments
        mlp: a list of layer dicts, as created by init_mlp
        X: an array of test data, where rows are samples
            and columns are features. feature dimension must
            match the input dimension of mlp.
        thresh: the decision threshold

    # Returns
        y_hat: a vector of predicted binary labels for X
    """
    # TODO: implement this
    return None
```

The `week_5.py` test script will use this to plot a classification map.

7.2 Play with the MLP hyperparameters

The `week_5.py` test script has command-line options for changing the training parameters and network structure. In particular, the `--layers` (or `-y`) option allows you to specify what hidden layer sizes are used. For example,

```
$ python week_5.py -y 6,8,4
```

will create a network with hidden layers containing 6, 8 and 4 neurons respectively.[2] Try varying the layer structure and seeing what effect it has on the boundary the network learns.

Also, the `--batch` (`-b`) option allows you to change the mini-batch size used during training. By default this is 1 (for stochastic gradient descent). What happens if you make it bigger? Does that have a qualitative effect on the outcome?

[2] The 2d input layer and 1d output are added automatically. Note that while the input has no weights, the output does, so this would be described as a "4-layer" network.