pjreddie / vision-hw4

Implement neural networks and run them on MNIST

15 commits | 1 branch | 0 releases | 1 contributor

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

pjreddie Update tryml.py — Latest commit b9aba51 on 11 May

data	hw4	8 months ago
figs	new line image	8 months ago
src	hw4	8 months ago
.gitignore	initial commit	8 months ago
Makefile	hw4	8 months ago
README.md	Update README.md	8 months ago
mnist.labels	Create mnist.labels	8 months ago
tryml.py	Update tryml.py	8 months ago
uwimg.py	hw4	8 months ago

README.md

CSE 455 Homework 4

Welcome friends,

It's time for neural networks!

To start out this homework, copy over your `process_image.c`, `filter_image.c`, `resize_image.c`, `harris_image.c`, `panorama_image.c`, and `flow_image.c` files from hw3 to the `src` directory in this homework. We will be continuing to build out your image library.

1. Implementing neural networks

We've added a bunch of new data structures and types to your image library as we begin to implement some machine learning algorithms. Check them out at line 122 in `src/image.h`.

1.1 Activation functions

An important part of machine learning, be it linear classifiers or neural networks, is the activation function you use. Fill in `void activate_matrix(matrix m, ACTIVATION a)` to modify `m` to be `f(m)` applied elementwise where the function `f` is given by what the activation `a` is.

Remember that for our softmax activation we will take e^x for every element x, but then we have to normalize each element by the sum as well. Each row in the matrix is a separate data point so we want to normalize over each data point separately.

1.2 Taking gradients

As we are calculating the partial derivatives for the weights of our model we have to remember to factor in the gradient from our activation function at every layer. We will have the derivative of the loss with respect to the output of a layer (after activation) and we need the derivative of the loss with respect to the input (before activation). To do that we take each element in our delta (the partial derivative) and multiply it by the gradient of our activation function.

Normally, to calculate `f'(x)` we would need to remember what the input to the function, `x`, was. However, all of our activation functions have the nice property that we can calculate the gradient `f'(x)` only with the output `f(x)`. This means we have to remember less stuff when running our model.

The gradient of a linear activation is just 1 everywhere. The gradient of our softmax will also be 1 everywhere because we will only use the softmax as our output with a cross-entropy loss function, for an explaination of why see here or here.

The gradient of our logistic function is discussed here:

```
f'(x) = f(x) * (1 - f(x))
```

I'll let you figure out on your own what `f'(x)` given `f(x)` is for RELU and Leaky RELU.

As discussed in the backpropagation slides, we just multiply our partial derivative by `f'(x)` to backpropagate through an activation function. Fill in `void gradient_matrix(matrix m, ACTIVATION a, matrix d)` to multiply the elements of `d` by the correct gradient information where `m` is the output of a layer that uses `a` as it's activation function.

1.3 Forward propagation

Now we can fill in how to forward-propagate information through a layer. First check out our layer struct:

```
typedef struct {
    matrix in;              // Saved input to a layer
    matrix w;               // Current weights for a layer
    matrix dw;              // Current weight updates
    matrix v;               // Past weight updates (for use with momentum)
    matrix out;             // Saved output from the layer
    ACTIVATION activation;  // Activation the layer uses
} layer;
```

During forward propagation we will do a few things. We'll multiply the input matrix by our weight matrix `w`. We'll apply the activation function `activation`. We'll also save the input and output matrices in the layer so that we can use them during backpropagation. But this is already done for you.

Fill in the TODO in `matrix forward_layer(layer *l, matrix in)`. You will need the matrix multiplication function `matrix matrix_mult_matrix(matrix a, matrix b)` provided by our matrix library.

Using matrix operations we can batch-process data. Each row in the input `in` is a separate data point and each row in the returned matrix is the result of passing that data point through the layer.

1.4 Backward propagation

We have to backpropagate error through our layer. We will be given `dL/dy`, the derivative of the loss with respect to the output of the layer, `y`. The output `y` is given by `y = f(xw)` where `x` is the input, `w` is our weights for that layer, and `f` is the activation function. What we want to calculate is `dL/dw` so we can update our weights and `dL/dx` which is the backpropagated loss to the previous layer. You'll be filling in `matrix backward_layer(layer *l, matrix delta)`

1.4.1 Gradient of activation function

First we need to calculate `dL/d(xw)` using the gradient of our activation function. Recall:

```
dL/d(xw) = dL/dy * dy/d(xw)
         = dL/dy * df(xw)/d(xw)
         = dL/dy * f'(xw)
```

Use your `void gradient_matrix(matrix m, ACTIVATION a, matrix d)` function to change delta from `dL/dy` to `dL/d(xw)`

1.4.2 Derivative of loss w.r.t. weights

Next we want to calculate the derivative with respect to our weights, `dL/dw`. Recall:

```
dL/dw = dL/d(xw) * d(xw)/dw
      = dL/d(xw) * x
```

but remember from the slides, to make the matrix dimensions work out right we acutally do the matrix operiation of `xt * dL/d(xw)` where `xt` is the transpose of the input matrix `x`.

In our layer we saved the input as `l->in`. Calculate `xt` using that and the matrix transpose function in our library, `matrix transpose_matrix(matrix m)`. Then calculate `dL/dw` and save it into `l->dw` (free the old one first to not leak memory!). We'll use this later when updating our weights.

1.4.3 Derivative of loss w.r.t. input

Next we want to calculate the derivative with respect to the input as well, `dL/dx`. Recall:

```
dL/dx = dL/d(xw) * d(xw)/dx
      = dL/d(xw) * w
```

again, we have to make the matrix dimensions line up so it actually ends up being `dL/d(xw) * wt` where `wt` is the transpose of our weights, `w`. Calculate `wt` and then calculate dL/dx. This is the matrix we will return.

1.5 Weight updates

After we've performed a round of forward and backward propagation we want to update our weights. We'll fill in `void update_layer(layer *l, double rate, double momentum, double decay)` to do just that.

Remember from class with momentum and decay our weight update rule is:

```
Δw_t = dL/dw_t - λw_t + mΔw_{t-1}
w_{t+1} = w_t + ηΔw_t
```

We'll be doing gradient ascent (the partial derivative component is positive) instead of descent because we'll flip the sign of our partial derivative when we calculate it at the output. We saved `dL/dw_t` as `l->dw` and by convention we'll use `l->v` to store the previous weight change `Δw_{t-1}`.

Calculate the current weight change as a weighted sum of `l->dw`, `l->w`, and `l->v`. The function `matrix axpy_matrix(double a, matrix x, matrix y)` will be useful here, it calculates the result of the operation `ax+y` where `a` is a scalar and `x` and `y` are matrices. Save the current weight change in `l->v` for next round (remember to free the current `l->v` first).

Finally, apply the weight change to your weights by adding a scaled amount of the change based on the learning rate.

1.6 Read through the rest of the code

Check out the remainder of the functions which string layers together to make a model, run the model forward and backward, update it, train it, etc. Notable highlights:

`layer make_layer(int input, int output, ACTIVATION activation)`

Makes a new layer that takes `input` number of inputs and produces `output` number of outputs. Note that we randomly initialize the weight vector, in this case with a uniform random distribution between `[-sqrt(2/input), sqrt(2/input)]`. The weights could be initialize with other random distributions but this one works pretty well. This is one of those secret tricks you just have to know from experience!

`double accuracy_model(model m, data d)`

Will be useful to test out our model once we are done training it.

`double cross_entropy_loss(matrix y, matrix p)`

Calculates the cross-entropy loss for the ground-truth labels `y` and our predictions `p`. Cross-entropy loss is just negative log-likelihood (we discussed log-likelihood in class for logistic regression) for multi-class classification. Since we added the negative sign it becomes a loss function that we want to minimize. Cross-entropy loss is given by:

```
L = Σ(-y_i log(p_i))
```

for a single data point, summing across the different classes. Cross-entropy loss is nice to use with our softmax activation because the partial derivatives are nice. If the output of our final layer uses a softmax: `y = σ(wx)` then:

```
dL/d(wx) = (y - truth)
```

During training, we'll just calculate `dL/dy` as `(y - truth)` and set the gradient of the softmax to be 1 everywhere to have the same effect, as discussed earlier. This avoids numerical instability with calculating the "true" `dL/dy` and `do(x)/dx` for cross-entropy loss and the softmax function independently and multiplying them together.

`void train_model(...)`

Our training code to implement SGD. First we get a random subset of the data using `data random_batch(data d, int n)`. Then we run our model forward and calculate the error we make `dL/dy`. Finally we backpropagate the error through the model and update the weights at every layer.

2. Experiments with MNIST

We'll be testing out your implementation on the MNIST dataset!

When there are questions during this section please answer them in the `tryml.py` file, you will submit this file in Canvas as well as `classifier.c`. Answers can be short, 1-2 sentences.

2.1 Getting the data

To run your model you'll need the dataset. The training images can be found here and the test images are here, I've preprocessed them for you into PNG format. To get the data you can run:

```
wget https://pjreddie.com/media/files/mnist_train.tar.gz
wget https://pjreddie.com/media/files/mnist_test.tar.gz
tar xzf mnist_train.tar.gz
tar xzf mnist_test.tar.gz
```

We'll also need a list of the images in our training and test set. To do this you can run:

```
find train -name \*.png > mnist.train
find test -name \*.png > mnist.test
```

Or something similar.

2.2 Train a linear softmax model

Check out `tryml.py` to see how we're actually going to run the machine learning code you wrote. There are a couple example models in here for softmax regression and neural networks. Run the file with: `python tryml.py` to train a softmax regression model on MNIST. You will see a bunch of numbers scrolling by, this is the loss calculated by the model for the current batch. Hopefully over time this loss goes down as the model improves.

After training our python script will calculate the accuracy the model gets on both the training dataset and a separate testing dataset.

2.2.1 Question

Why might we be interested in both training accuracy and testing accuracy? What do these two numbers tell us about our current model?

2.2.2 Question

Try varying the model parameter for learning rate to different powers of 10 (i.e. 10^1, 10^0, 10^-1, 10^-2, 10^-3) and training the model. What patterns do you see and how does the choice of learning rate affect both the loss during training and the final model accuracy?

2.2.3 Question

Try varying the parameter for weight decay to different powers of 10: (10^0, 10^-1, 10^-2, 10^-3, 10^-4, 10^-5). How does weight decay affect the final model training and test accuracy?

2.3 Train a neural network

Now change the training code to use the neural network model instead of the softmax model.

2.3.1 Question

Currently the model uses a logistic activation for the first layer. Try using a the different activation functions we programmed. How well do they perform? What's best?

2.3.2 Question

Using the same activation, find the best (power of 10) learning rate for your model. What is the training accuracy and testing accuracy?

2.3.3 Question

Right now the regularization parameter `decay` is set to 0. Try adding some decay to your model. What happens, does it help? Why or why not may this be?

2.3.4 Question

Modify your model so it has 3 layers instead of two. The layers should be `inputs -> 64`, `64 -> 32`, and `32 -> outputs`. Also modify your model to train for 3000 iterations instead of 1000. Look at the training and testing accuracy for different values of decay (powers of 10, 10^-4 -> 10^0). Which is best? Why?

3. Training on CIFAR

The CIFAR-10 dataset is meant to be similar to MNIST but much more challenging. Using the same model we just created, let's try training on CIFAR.

3.1 Get the data

We have to do a similar process as last time, getting the data and creating files to hold the paths. Run:

```
wget http://pjreddie.com/media/files/cifar.tgz
tar xzf cifar.tgz
find cifar/train -name \*.png > cifar.train
find cifar/test -name \*.png > cifar.test
```

Notice that the file of possible labels can be found in `cifar/labels.txt`

3.2 Train on CIFAR

Modify `tryml.py` to use CIFAR. This should mostly involve swapping out references to `mnist` with `cifar` during dataset loading. Then try training the network. You may have to fiddle with the learning rate to get it to train well.

3.2.1 Question

How well does your network perform on the CIFAR dataset?

4. Turn it in

Turn in your `classifier.c` and `tryml.py` on canvas under Assignment 4.