1 Task

Recreate a simple neural network for supervised classification! The following needs to be supported:

- Load an image dataset. The dataset is specified in a textfile of form `filename label`
- Use object-oriented programming to implement layers
- The following layers need to be supported:
  - Data loading layer
  - Fully-Connected layer
  - ReLU, Sigmoid, Tanh activation layer
  - Softmax layer
  - Cross-Entropy Classification loss
  - Accuracy layer
- The network needs to support inference and learning (forward & backward operations)

1.1 Constraints

Object-oriented: You need to use the classes `Layer` in `layer.hpp` and `Datum` in `datum.hpp`. All network layers should inherit `Layer` and implement the virtual functions.

Parameterized layers need to initialize their weights (see Section 2.1.2). This should be done in the `SetUp()` function.

2 Supporting material

2.1 Neural Networks

Generally, a neural network works as follows: A *datum*, e.g., an image, is read. This datum is then *forwarded* through the network layer-by-layer. The operation in each layer differs with its type.

After a full forward, a loss is typically computed. Here we use the Cross-Entropy loss (see Section 2.5.1). The loss resembles how badly the network works, i.e., the error.

Given this loss, we start the *backpropagation* phase. Starting from the last layer (typically the loss), we go back layer by layer and update the gradient. Layers that are parameterized (Fully-Connected layer) need to update their weights. This is done only *after* the backpropagation is finished. In the meantime, the appropriate gradients are stored in temporary memory.

The effect of the gradient update is determined by the *learning rate*. It is simply multiplied on the gradient. Here, we can assume a constant value between $0.01 \leq \alpha \leq 0.001$.

2.1.1 Gradient computation

Say we have a fully connected layer followed by an activation. We are given the loss (gradient) coming from higher layers. The first step is to compute the gradient with respect to *before* the activation function. Let $\mathbf{g}_{top} \in \mathbb{R}^{out}$ be the gradient coming from the top, and let the activation function be a sigmoid $\sigma(x)$. Then, the new gradient is:

$$\mathbf{g}_{act} = \mathbf{g}_{top} \cdot \frac{\partial \sigma(x)}{\partial x} = \mathbf{g}_{top} \cdot \sigma(x) \cdot (1 - \sigma(x))$$

Now, we can compute the gradient at the fully connected layer. In a fully connected layer, we compute the product of some input $\mathbf{x} \in \mathbb{R}^{in}$ and weights $\mathbf{W} \in \mathbb{R}^{out \times in}$. We also add a bias $\mathbf{b} \in \mathbb{R}^{out}$:

$$\mathbf{y} = \mathbf{W} \cdot \mathbf{x} + \mathbf{b}, \quad \mathbf{y} \in \mathbb{R}^{out}$$

In other words, for each output unit, we first compute a dot-product

$$y_i = \mathbf{w_i} \cdot \mathbf{x} + b_i.$$

We begin by computing the gradient for the elements of $\mathbf{w_i}$. Because the derivative of $\mathbf{y}$ with regard to $\mathbf{w}$ is:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{w}_i} = \mathbf{1} \cdot \mathbf{x} + 0$$

we can compute the gradient for the weights very easily:

$$\frac{\partial \mathbf{g}_{act}}{\partial w_{ij}} = \alpha \cdot g_{act} \cdot x_j,$$

where α is the *learning rate* mentioned previously.

As mentioned above, we do not update the weights right away. This is because it is useful to average the gradients of N input examples to get more stable learning. In that case you must use the same weights for each input example. This idea (called *batch learning*) is optional for your task.

We already computed the gradient for the weights. However, we also need to compute the new gradient with regard to before the fully-connected layer, i.e., the input. Again, the derivative of $\mathbf{y}$ with regard to $\mathbf{x}$ this time:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{W} \cdot \mathbf{1} + 0$$

Therefore, we compute the gradient g_{in} as:

$$\mathbf{g}_{in}^{(j)} = \frac{\partial \mathbf{g}_{act}}{\partial x_j} = \sum_{i=1}^{out} \mathbf{g}_{act}^{(i)} \cdot w_{ij}$$

That is, the sum over the i-th element of $\mathbf{g}_{act}$ times the weight w_{ij}.

2.1.2 Initialization of parameters

We need to provide initial values for the weights $\mathbf{W}$ and the bias $\mathbf{b}$. If this is not done, the network will **not** learn! Several approaches exist:

Gaussian Initialize each value by drawing a value of $\mathcal{N}(\mu, \sigma)$. Typical values could be $\mu = 0, \sigma = 0.01$. This can be implemented easily using the `random` header in C++11.

Uniform Typically, the values are sampled within $\pm\frac{2}{n_{in}+n_{out}}$, where n_{in} the number of incoming connections, and n_{out} the number of outgoing connections.

Constant This method is used for the *bias* only. Typical values range between 0 and 1.

2.2 Activation functions

2.2.1 ReLU

$$ReLU(x) = \begin{cases} x & x \geq 0 \\ 0 & otherwise \end{cases}$$

We define the gradient at x=0 to be zero.

2.2.2 Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\left(1-\sigma(x)\right)$$

2.2.3 TanH

$$\tanh(x) = \frac{e^{z}-e^{-z}}{1+e^{-z}} = 2\sigma(x) - 1$$

$$\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh^2(x)$$

2.3 Fully Connected

This layer computes the dot-product

$$\mathbf{y} = \mathbf{W} \cdot \mathbf{x} + \mathbf{b}.$$

For details, see gradient computation (Sec. 2.1.1).

2.4 Softmax

The softmax function is not used as an activation function, but in order to compute a probability distribution (where all probabilities sum to 1). In addition, simply said, it makes the strongest element of a vector go against one, while pushing all other elements towards zero. The definition is:

$$\sigma(x_i) = \frac{e^{x_i}}{\sum\limits_j e^{x_j}}$$

The gradient is usually computed directly with the loss, see Sec. 2.5.1.

2.5 Loss

2.5.1 Cross-Entropy

The definition of the cross-entropy loss for N input samples is:

$$L(w) = -\frac{1}{N}\sum_{n=1}^{N}\left(\hat{y}_n \log y_n + (1-\hat{y}_n)\log(1-y_n)\right),$$

where $\hat{y}_n$ are the target probabilities.

We compute the loss between a label and the softmax output. The softmax output is a vector, so we encode the (scalar) label in a *one-hot* representation – if the label is '2', and we discriminate between four classes, then the target vector is $[0,0,1,0]$.

Only considering the non-zero entries, the target probability $\hat{y}_n = 1$, and therefore we can simplify the loss considerably:

$$L(w) = -\frac{1}{N}\sum_{n=1}^{N}\log y_n$$

The gradient of Cross-Entropy and Softmax combined is[1]:

$$\frac{\partial L}{\partial x_i} = \underbrace{y_i}_{\text{output of softmax}} - \underbrace{\hat{y}_i}_{\in\{0,1\}}$$

Example: if after softmax we have $\mathbf{y} = [0.25, 0.25, 0.5]$ and the label is $\hat{\mathbf{y}} = [0,0,1]$, then the gradient is $\mathbf{g} = [0.25, 0.25, -0.5]$.

2.6 Accuracy

This layer computes the accuracy. For that, simply check if the maximum of the network output coincides with the '1' in the target vector.

[1] `http://peterroelants.github.io/posts/neural_network_implementation_intermezzo02/`

Example:

- Network output is $[0.1, 0.3, 0.6]$ and target value is $[0, 0, 1]$: Correct classification.
- Network output is $[0.1, 0.3, 0.6]$ and target value is $[0, 1, 0]$: Incorrect classification.

The accuracy is then the average over all test-set examples.