Deep Learning Acceleration

嵌入式多核心系統與軟體期末專題

Overview

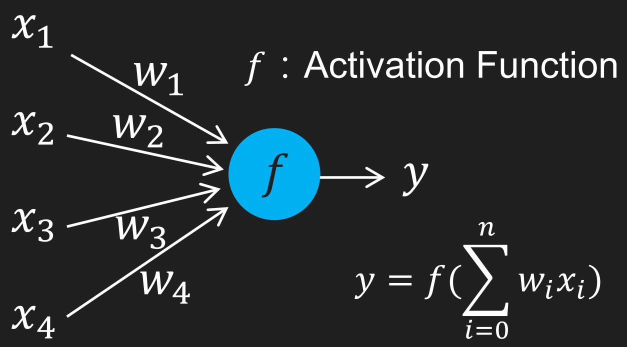
Deep learning requires large computations. By using the GPU computing power, we can reduce the training time significantly. This is because the basic architecture of neural networks is based on matrix operations and GPU is a hardware platform that's been optimized for this.

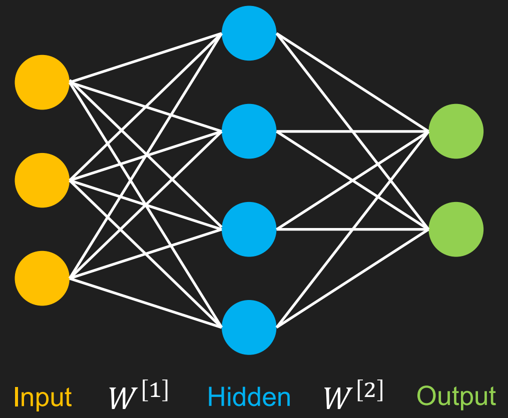
Components

- Fully-connected Layer
- Forward Propagation
- Gradient Backward Propagation
- Activation Function (Sigmoid / ReLU)
- Softmax Layer
- Loss Function
- Convolution Layer (depends on the progress)

Review of Deep Learning

- $h = \sigma(z) = \sigma\left(W^{T[1]}x + b^{[1]}\right)$
- $\hat{y} = softmax(o) = softmax(W^{T[2]}h + b^{[2]})$
- $Loss = J_{CE}(\hat{y}, y) = -\sum_{i=0}^{m} y^{(i)} \log(\hat{y}^{(i)})$
- $\frac{\partial L}{\partial z} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial h}\frac{\partial h}{\partial z} = \delta_1 W^{T[2]}\frac{\partial h}{\partial z} = \delta_1 W^{T[2]}\sigma'(z)$
- $W^{[l+1]} = W^{[l]} - \mu \nabla W^{[l]}$, $b^{[l+1]} = b^{[l]} - \mu \nabla b^{[l]}$



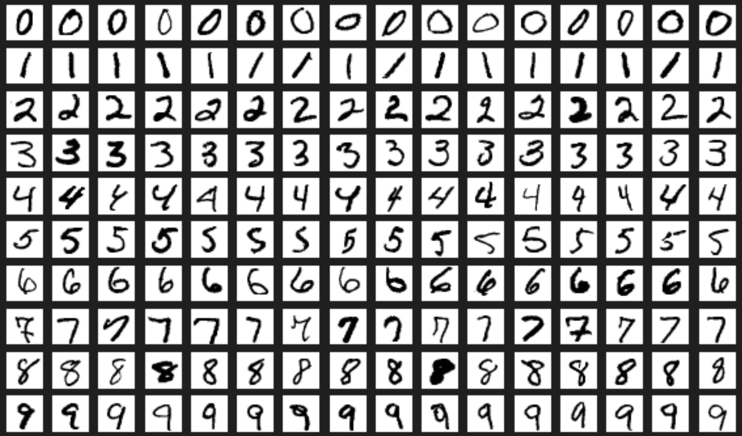


Libraries

- OpenCV: Computer vision library for image processing
- CUDA: Parallel computing platform and programming model
- cuBLAS: GPU-accelerated library for basic linear algebra subroutines
- cuDNN: GPU-accelerated library for deep neural networks

Goal

- Train a neural network on the MNIST dataset
- Recognize hand-written digits



Implementation: Layer Operations

- Forward propagation
- Backward propagation
- Update the weights and biases (Dense layer)
- Obtain the loss (Softmax layer)

cuBLAS API

- `cublasSaxpy`: $\vec{y} = \alpha\vec{x} + \vec{y}$
- `cublasSgemm`: $C = \alpha A \cdot B + \beta C$
- `cublasSgemv`: $\vec{y} = \alpha A \cdot \vec{x} + \beta\vec{y}$
- `cublasSscal`: $\vec{x} = \alpha \cdot \vec{x}$

Implementation: Dense Layer

Forward

$$y = W^T \cdot x + b$$

```
Tensor<float> *Dense::forward(Tensor<float> *input) {
    // y = W^T * x (without biases)
    cublasSgemm(cuda_->cublas(),
                CUBLAS_OP_T, CUBLAS_OP_N,
                output_size_, batch_size_, input_size_,
                &cuda_->one,
                weights_->cuda(), input_size_,
                input_->cuda(), input_size_,
                &cuda_->zero,
                output_->cuda(), output_size_);

    // y += b * one_vec^T
    cublasSgemm(cuda_->cublas(),
                CUBLAS_OP_N, CUBLAS_OP_N,
                output_size_, batch_size_, 1,
                &cuda_->one,
                biases_->cuda(), output_size_,
                d_one_vec, 1,
                &cuda_->one,
                output_->cuda(), output_size_);

    return output_;
}
```

Implementation: Dense Layer

Backward

$$dx = W \cdot dy$$

$$dw = x \cdot dy^T$$

$$db = dy \cdot \vec{1}$$

```
Tensor<float> *Dense::backward(Tensor<float> *grad_output) {
    // dx = W * dy
    if (!gradient_stop_)
        cublasSgemm(cuda_->cublas(),
                    CUBLAS_OP_N, CUBLAS_OP_N,
                    input_size_, batch_size_, output_size_,
                    &cuda_->one,
                    weights_->cuda(), input_size_,
                    grad_output_->cuda(), output_size_,
                    &cuda_->zero,
                    grad_input_->cuda(), input_size_);

    // dw = x * dy^T
    cublasSgemm(cuda_->cublas(),
                CUBLAS_OP_N, CUBLAS_OP_T,
                input_size_, output_size_, batch_size_,
                &cuda_->one,
                input_->cuda(), input_size_,
                grad_output_->cuda(), output_size_,
                &cuda_->zero,
                grad_weights_->cuda(), input_size_);

    // db = dy * one_vec
    cublasSgemv(cuda_->cublas(),
                CUBLAS_OP_N,
```

Implementation: Dense Layer

Update weights

$$W^{[l+1]} = W^{[l]} - \mu \nabla W^{[l]}$$

Update biases

$$b^{[l+1]} = b^{[l]} - \mu \nabla b^{[l]}$$

```
void Layer::update_weights_biases(float learning_rate) {
    float eps = -1.f * learning_rate;
    if (weights_ != nullptr && grad_weights_ != nullptr) {
        // w = eps * dw + w
        cublasSaxpy(cuda_->cuBLAS(),
                    weights_->len(),
                    &eps,
                    grad_weights_->cuda(), 1,
                    weights_->cuda(), 1);
    }

    if (biases_ != nullptr && grad_biases_ != nullptr) {
        // b = eps * db + b
        cublasSaxpy(cuda_->cublas(),
                    biases_->len(),
                    &eps,
                    grad_biases_->cuda(), 1,
                    biases_->cuda(), 1);
    }
}
```

Implementation: Activation Layer

Forward

$$y = ReLU(x)$$

Backward

$$dx = ReLU'(dy)$$

```
Tensor<float> *Activation::forward(Tensor<float> *input) {
    cudnnActivationForward(cuda_->cudnn(),
                           act_desc_,
                           &cuda_->one,
                           input_desc_,
                           input->cuda(),
                           &cuda_->zero,
                           output_desc_,
                           output_->cuda());
    return output_;
}
```

```
Tensor<float> *Activation::backward(Tensor<float> *grad_output) {
    cudnnActivationBackward(cuda_->cudnn(),
                            act_desc_,
                            &cuda_->one,
                            output_desc_, output_->cuda(),
                            output_desc_, grad_output->cuda(),
                            input_desc_, input_->cuda(),
                            &cuda_->zero,
                            input_desc_, grad_input_->cuda());
    return grad_input_;
}
```

Implementation: Softmax Layer

Forward

$$\hat{y} = softmax(x)$$

Backward

$$\nabla J_{CE}(\hat{y}, y) = \hat{y} - y$$

```
Tensor<float> *Softmax::forward(Tensor<float> *input) {
    cudnnSoftmaxForward(cuda_->cudnn(),
        CUDNN_SOFTMAX_ACCURATE, CUDNN_SOFTMAX_MODE_CHANNEL,
        &cuda_->one, input_desc_, input->cuda(),
        &cuda_->zero, output_desc_, output_->cuda());
    return output_;
}
```

```
Tensor<float> *Softmax::backward(Tensor<float> *target) {
    // Set gradient input as predict
    cudaMemcpyAsync(grad_input_->cuda(),
                    output_->cuda(), output_->buf_size(),
                    cudaMemcpyDeviceToDevice);

    // Set gradient input = predict - target
    cublasSaxpy(cuda_->cublas(), target->len(),
                &cuda_->minus_one, target->cuda(), 1,
                grad_input_->cuda(), 1);

    // Normalize the gradient output by the batch size
    int grad_output_size =
        target->n() * target->c() * target->h() * target->w();
    float scale = 1.f / static_cast<float>(target->n());
    cublasSscal(cuda_->cublas(), grad_output_size, &scale,
                grad_input_->cuda(), 1);
```

Implementation: Loss Function

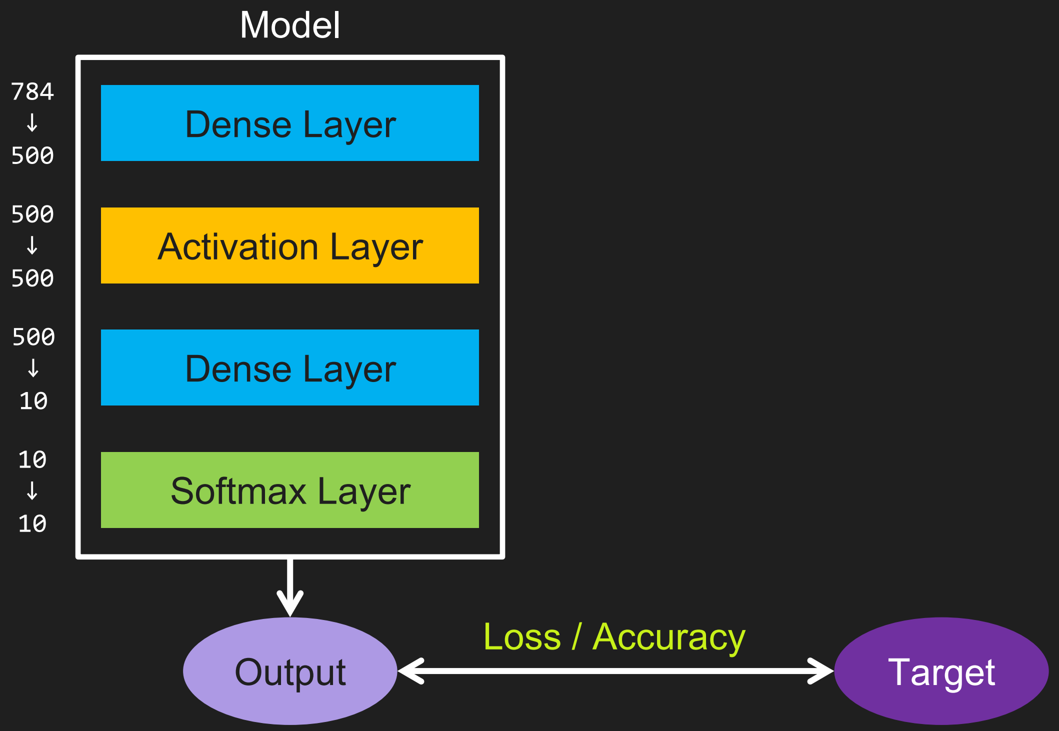
- Use this as an indicator of the training (Optional)
- Obtain the loss from outputs and cumulate them using a kernel function

$$Loss = J_{CE}(\hat{y}, y) = -\sum_{i=0}^{m} y^{(i)} \log\left(\hat{y}^{(i)}\right) = -\sum_{i=0}^{m} y^{(i)} \log\left(softmax\left(o^{(i)}\right)\right)$$

```
int num_blocks = std::min(num_blocks_per_sm * num_sms,
                          (target->size() + BLOCK_DIM_1D - 1) / BLOCK_DIM_1D);
softmax_loss_kernel<<<num_blocks, BLOCK_DIM_1D, BLOCK_DIM_1D * sizeof(float), nullptr>>>
        (d_loss_, predict->cuda(), target->cuda(), d_workspace_, batch_size, num_outputs);
```

```
__global__ void softmax_loss_kernel(float *reduced_loss, float *predict, const float *target,
                                    float *workspace, int batch_size, int num_outputs) {
    int batch_idx = (int) (blockDim.x * blockIdx.x + threadIdx.x);
    extern __shared__ float s_data[];
    float loss = 0.f;

    // Each thread calculates entropy for each data and accumulates them to shared memory
    for (int c = 0; c < num_outputs; c++)
        loss += target[batch_idx * num_outputs + c] * logf(predict[batch_idx * num_outputs + c]);
    workspace[batch_idx] = -loss;

    // Then, we do the reduction on the result to calculate loss using 1 thread block
    if (blockIdx.x > 0) return;

    // Cumulate workspace data
    s_data[threadIdx.x] = 0.f;
    for (int i = 0; i < batch_size; i += (int) blockDim.x)
        s_data[threadIdx.x] += workspace[threadIdx.x + i];
    __syncthreads();

    // Reduction
    for (unsigned int stride = blockDim.x / 2; stride > 0; stride >>= 1) {
        if (threadIdx.x + stride < batch_size)
            s_data[threadIdx.x] += s_data[threadIdx.x + stride];
        __syncthreads();
    }

    if (threadIdx.x == 0) reduced_loss[blockIdx.x] = s_data[0];
}
```

Implementation: Network



Implementation: Training

- Create the model layers and apply the layer operations

```cpp
// Initializing model
Network model;
model.addLayer(new Dense("Dense 1", 500));
model.addLayer(new Activation("ReLU",
               CUDNN_ACTIVATION_RELU));
model.addLayer(new Dense("Dense 2", 10));
model.addLayer(new Softmax("Softmax"));
model.cuda();

if (loadPretrain)
    model.loadPretrain();

...
```

```cpp
while (step < trainStepNum) {
    // Updating shared buffer contents
    trainData->to(cuda);
    getTarget->to(cuda);

    // Forward propagation
    model.forward(trainData);

    // Backward propagation
    model.backward(getTarget);

    // Updating learning rate, weights and biases
    learningRate *= 1.f / (1.f + lrDecay * step);
    model.update((float) learningRate);

    // Fetching the next data
    step = trainDataLoader.next();
}
```

Implementation: Inference

- Take an image as input and recognize the hand-written digit

```
// Loading image
MNIST dataLoader = MNIST(“.”);
dataLoader.loadImage(“5.jpg”);

// Predict
Tensor<float> *image = dataLoader.getData();
image->to(cuda);
model.predict(image);
```

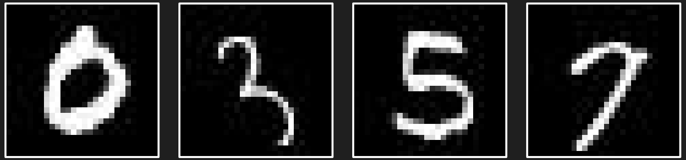
Experiment

- Model layers: **Dense → ReLU → Dense → Softmax**
- Batch size: 256
- Number of steps: 1600
- Learning rate: 0.02
- Learning rate decay: 0.00005

Results

- Training accuracy: 93.3%
- Testing accuracy: **80.9%**

```
[TRAINING]
step:  200 loss: 0.538 accuracy: 82.170%
step:  400 loss: 0.476 accuracy: 93.158%
step:  600 loss: 0.481 accuracy: 93.354%
step:  800 loss: 0.517 accuracy: 93.371%
step: 1000 loss: 0.521 accuracy: 93.389%
step: 1200 loss: 0.465 accuracy: 93.367%
step: 1400 loss: 0.483 accuracy: 93.352%
step: 1600 loss: 0.480 accuracy: 93.355%
```



Conclusion

- The network achieved 93% accuracy from the training dataset
- Testing accuracy is 81%, which is relative lower than the training result
- A gap in accuracy between training and inference
- Possible reason: fully-connected layer not consider the regional info
- Improvement: add a convolutional layer

Reference

- Jason Sanders, Edward Kandrot. (2010). CUDA by Example: An Introduction to General-Purpose GPU Programming. USA: Addison-Wesley Professional.
- Jaegeun Han, Bharatkumar Sharma. (2019). Learn CUDA Programming: A beginner's guide to GPU programming and parallel computing with CUDA 10.x and C/C++. UK: Packt Publishing Ltd.
- NVIDIA. (2023). CUDA C++ Programming Guide. USA: NVIDIA Corporation.

Thanks