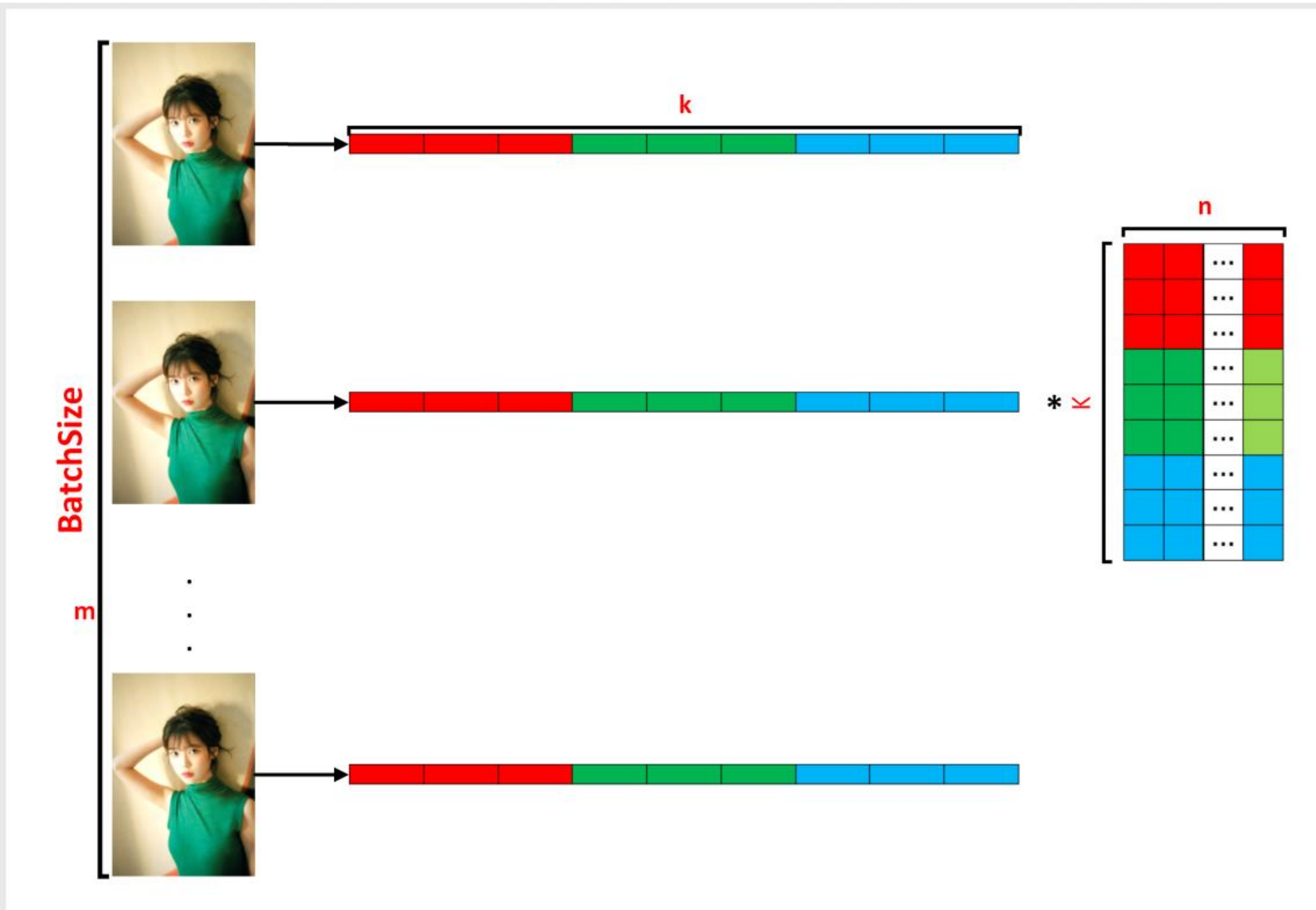
源码图解02-全连接层

2018年11月23日　16:14

connected_layer.c

正向传播

```
void forward_connected_layer(layer l, network net)

int m = l.batch;
int k = l.inputs;
int n = l.outputs;
float *a = net.input;
float *b = l.weights;
float *c = l.output;
```



B-n x k

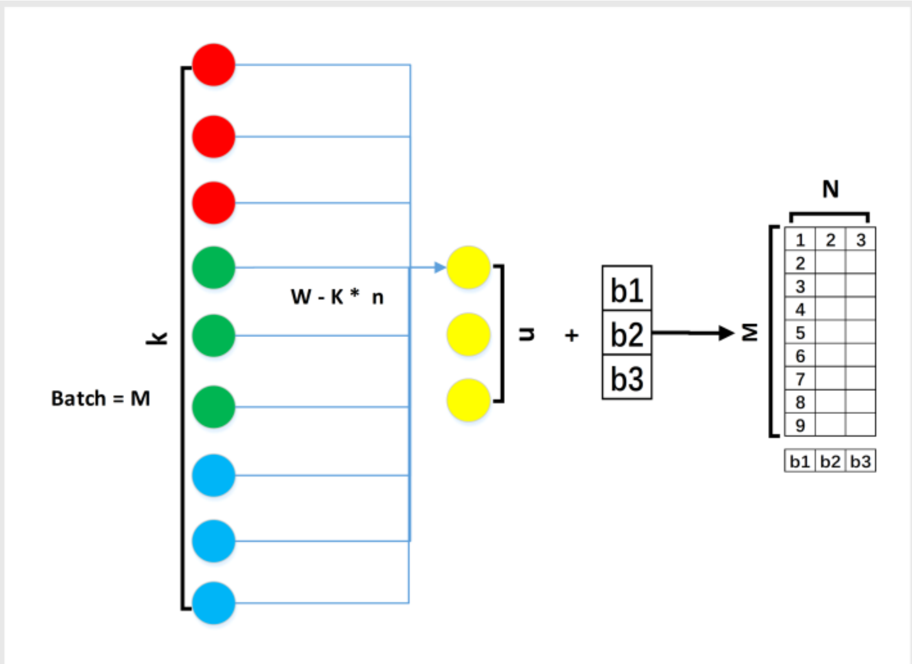
C-m x n = A-m x k * B'- k x n + C- m x n

gemm(0,1,m,n,k,1,a,k,b,k,1,c,n);

C-m x n M张图片，N个类别，一行元素代表每一张图片

在每个类别的概率值。每一列是一个神经元的输出值，共享一个bias.

```
add_bias(l.output, l.biases, l.batch, l.outputs, 1);

void add_bias(float *output, float *biases, int batch, int n, int size)
{
    int i,j,b;
    for(b = 0; b < batch; ++b){
        for(i = 0; i < n; ++i){
            for(j = 0; j < size; ++j){
                output[(b*n + i)*size + j] += biases[i];
            }
        }
    }
}
```




```
activate_array(l.output, l.outputs*l.batch, l.activation);
```

反向传播：

//输入： l-当前层对象，封装了当前层的各项属性

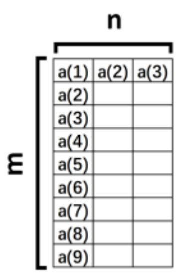
//　　　 net-整个网络，封装了整个网络结构

```
void backward_connected_layer(layer l, network net)
```

//反向传播操作步骤：1）计算当前层的梯度 （梯度传入）

//2）更新当前层W,b 3）计算下一层的梯度（梯度传出）

//梯度传入当前层，先对激活函数求梯度
//存入l.delta



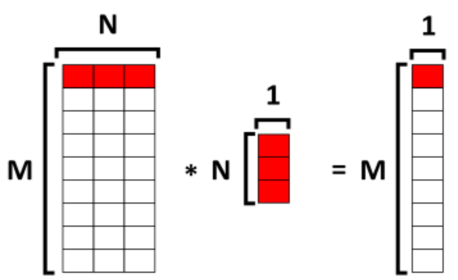
m-l.batch **n**-l.outputs

```
gradient_array(l.output, l.outputs*l.batch, l.activation, l.delta);

void gradient_array(const float *x, const int n, const ACTIVATION a, float *delta)
{
    int i;
    for(i = 0; i < n; ++i){
        delta[i] *= gradient(x[i], a);
    }
}

float gradient(float x, ACTIVATION a)
{
    switch(a){
        case LINEAR:
            return linear_gradient(x);
        case LOGISTIC:
            return logistic_gradient(x);
        case LOGGY:
            return loggy_gradient(x);
        case RELU:
            return relu_gradient(x);
        case ELU:
            return elu_gradient(x);
```

```
        case RELIE:
            return relie_gradient(x);
        case RAMP:
            return ramp_gradient(x);
        case LEAKY:
            return leaky_gradient(x);
        case TANH:
            return tanh_gradient(x);
        case PLSE:
            return plse_gradient(x);
        case STAIR:
            return stair_gradient(x);
        case HARDTAN:
            return hardtan_gradient(x);
        case LHTAN:
            return lhtan_gradient(x);
    }
    return 0;
}
```

//更新bias

```
backward_bias(l.bias_updates, l.delta, l.batch, l.
outputs, 1);
```



$$\frac{\partial loss}{\partial b} = \sum_{i}^{n} \frac{\partial loss}{\partial layerOutput_i} \cdot \frac{\partial layerOutput_i}{\partial b} = \sum_{i}^{n} layerOutput_i'$$

L.delta

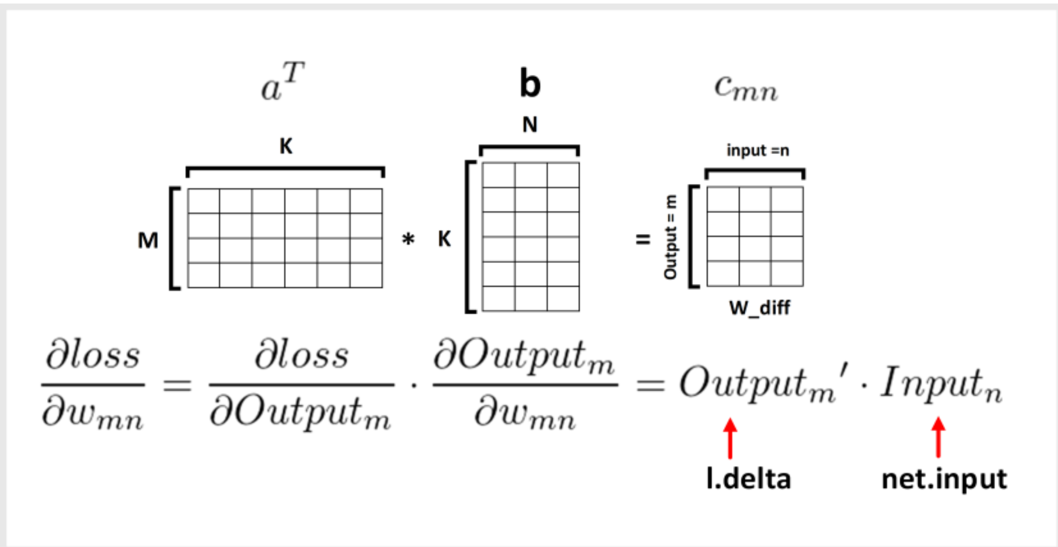
```
void backward_bias(float *bias_updates, float *delt
a, int batch, int n, int size)
{
    int i,b;
    for(b = 0; b < batch; ++b){
        for(i = 0; i < n; ++i){
            bias_updates[i] +=
```

```
                    sum_array(delta+size*(i+b*n), si
                    ze);
        }
    }
}
```

//更新W

```
    int m = l.outputs;
    int k = l.batch;
    int n = l.inputs;
    float *a = l.delta;
    float *b = net.input;
    float *c = l.weight_updates;
    gemm(1,0,m,n,k,1,a,m,b,n,1,c,n);
```

源码图解02-全连接层 - 绘图



//传出梯度

```
    m = l.batch;
    k = l.outputs;
    n = l.inputs;

    a = l.delta;
    b = l.weights;
    c = net.delta;
```

$$\frac{\partial loss}{\partial layerInput_j} = \sum_i^n \frac{\partial loss}{\partial layerOutput_i} \cdot \frac{\partial layerOutput_i}{\partial layerInput_j} = \sum_i^n layerOutput_i' \cdot w_{ij}$$

$$a_{mk} \cdot b_{kn} = c_{mn}$$

```
if(c) gemm(0,0,m,n,k,1,a,k,b,n,1,c,n);
```