Artificial Neural Networks

2nd Assignment

Saeid Cheshmi- 98222027

Exercise 1

For describing backpropagation in CNN, first we define cross-correlation and forward pass in CNN then, go through how backpropagation works.

With an input image I and kernel K with dimension of k1×k2, cross-correlation defined as below:

$$(I \otimes K)_{ij} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} I(i+m, j+n) K(m,n)$$

Convolution is the same as cross-correlation with a flipped kernel i.e: for a kernel K where K(−m,−n)==K(m,n).

Forward pass

To perform a convolution operation, the kernel goes across the input feature map in equal and finite strides. At each location, the product between each element of the kernel and the input feature map element it overlaps is computed and the results summed up to obtain the output at that current location. This procedure is repeated using different kernels to form as many outputs feature maps as desired.

Backward pass

In the classical backpropagation algorithm, the weights are changed according to the gradient descent direction of an error surface E.

In backpropagation two updates preform, weights and deltas. We begin with weights; we are going to compute derivative of E w.r.t ${w^l}_{m,n}$ which can be interpreted as how change in a single pixel ${w^l}_{m,n}$ in the weight kernel affects the loss function E. During forward propagation, the convolution operation ensures that the yellow pixel ${w^l}_{m,n}$in the weight kernel contributes to all the products. This means that pixel t ${w^l}_{m,n}$ will eventually affect all the elements in the output feature map.

The gradient component for the individual weights can be calculated by applying the chain rule in this way:

$$\frac{\partial E}{\partial w^l_{m',n'}} = \sum_{i=0}^{H-k_1} \sum_{j=0}^{W-k_2} \frac{\partial E}{\partial x^l_{i,j}} \frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}}$$
$$= \sum_{i=0}^{H-k_1} \sum_{j=0}^{W-k_2} \delta^l_{i,j} \frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}}$$

In above equation $x^l{}_{i,j}$ is equivalent to be

$$\sum_m \sum_n w^l_{m,n} o^{l-1}_{i+m,j+n} + b^l$$

and we can write the equation as below:

$$\frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}} = \frac{\partial}{\partial w^l_{m',n'}} \left(\sum_m \sum_n w^l_{m,n} o^{l-1}_{i+m,j+n} + b^l \right)$$

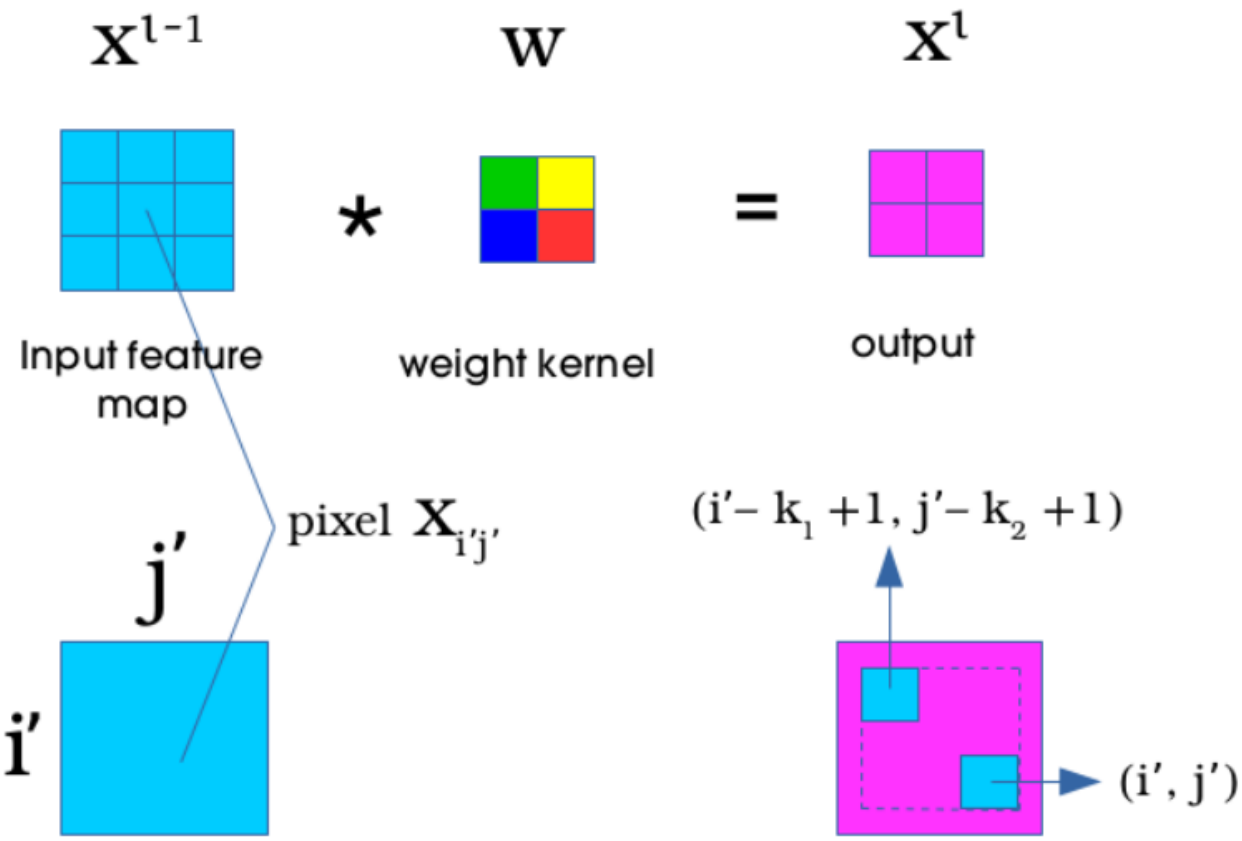
In above equation after taking the partial derivatives for all the components results in zero values for all except the components where m=m′ and n=n′ in:

$$w^l_{m,n} o^{l-1}_{i+m,j+n}$$

We can summarize the equation as below:

$$\frac{\partial E}{\partial w^l_{m',n'}} = \sum_{i=0}^{H-k_1} \sum_{j=0}^{W-k_2} \delta^l_{i,j} o^{l-1}_{i+m',j+n'}$$

The dual summation in above equation is as a result of weight sharing in the network, same weight kernel is taken over all of the input feature map during convolution. The summations represents a collection of all the gradients δi,j coming from all the outputs in layer l. Obtaining gradients w.r.t to the filter maps, we have a cross-correlation which is transformed to a convolution by “flipping” the delta matrix the same way we flipped the filters during the forward propagation.



In above diagram we can see that region in the output affected by pixel xi′,j′ from the input is the region in the output bounded by the dashed lines where the top left corner pixel is given by (i′−k1+1,j′−k2+1) and the bottom right corner pixel is given by (i′,j′).

By using chain rule and introducing sums give us the following equation:

$$\frac{\partial E}{\partial x_{i',j'}^{l}} = \sum_{i,j \in Q} \frac{\partial E}{\partial x_{Q}^{l+1}} \frac{\partial x_{Q}^{l+1}}{\partial x_{i',j'}^{l}}$$

$$= \sum_{i,j \in Q} \delta_{Q}^{l+1} \frac{\partial x_{Q}^{l+1}}{\partial x_{i',j'}^{l}}$$

Q in the summation above represents the output region bounded by dashed lines and is composed of pixels in the output that are affected by the single pixel xi′,j′ in the input feature map.

For backpropagation, we use of the flipped kernel and as a result we will now have a convolution that is expressed as a cross-correlation with a flipped kernel:

$$\frac{\partial E}{\partial x_{i',j'}^{l}} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} \delta_{i'-m,j'-n}^{l+1} w_{m,n}^{l+1} f'\left(x_{i',j'}^{l}\right)$$

$$= \mathrm{rot}_{180^\circ} \left\{ \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} \delta_{i'+m,j'+n}^{l+1} w_{m,n}^{l+1} \right\} f'\left(x_{i',j'}^{l}\right)$$

$$= \delta_{i',j'}^{l+1} * \mathrm{rot}_{180^\circ} \left\{ w_{m,n}^{l+1} \right\} f'\left(x_{i',j'}^{l}\right)$$

Exercise 2

We explain this phenomenon with some examples. One way to initialize neural networks weights is zero initialization. initializing the weight and bias to be zero will surely bring us to the dead neuron problem. In each iteration of gradient descent, we will find gradient for the previous layer by multiplying the existing weight with a gradient obtained by backpropagation from the next layer. If initial weight is zero multiplying it by any gradient will set the gradient to zero. Due to zero gradient, any weight won't change, and network doesn't learn patterns. But if we set bias to a positive value, the weights will change because activation function, say ReLU, produces non-zero output but they're going to change in an undesirable way. After that, every neuron in the same layer has same behavior and will be ended up having the same weight. This phenomenon is called symmetric problem.

Same problem will occur if we set any weight in each layer to constant value and bias to zero. One way to address this issue is random weight initialization. This allows us to break symmetry. But this isn't enough to train deep neural network. Large wights lead to exploding gradient, while small weights lead to vanishing gradient. Also, every neuron will have a different output range since we give it weight without specific range. To address this problem, we can initialize the weights with random normal distribution. By initializing weights with random normal distribution, we can set mean and std to give a range to weights. There is also another way to initialize a neural network, **Xavier Initialization**, in this way we can control the variance of output. We want the output of a layer which is produced by neurons to follow same distribution. **Xavier Initialization** does this for us.

Exercise 3

Pooling layers are one of building blocks of CNN. where convolutional layers extract features from images, pooling layers combine the features learned by CNNs. It gradually shrinks the spatial dimension to minimize the number of parameters and computations in the network.

The feature map produced by the filters is location dependent. It means that feature map records the precise positions of features in the input. What pooling makes the CNN invariant to translations? Pooling layers have two main advantages, 1) reduce the number of parameters, thus it controls the overfitting and decreases computation cost, 2) make model invariant to certain distortion, as mentioned above.

There are two main types of pooling, average pooling and max pooling. Due to its focus on extreme values, max pooling is attentive to the more prominent features and edges in the image. Average pooling, on the other hand, creates a smoother feature map because it produces averages instead of selecting the extreme values. Max pooling is applied more often because it is generally better at identifying prominent features.

There are some drawbacks when we use pooling layers. If filter size is too big some spatial information may be lost, especially when we use max pooling. If max pooling is applied an image where its background is light, it eliminates the foreground features and only return the background.

As a result, pooling is helpful when we have an image classification task because you just need to detect to presence of a certain object in an image. Also, pooling layers use larger stride than convolution layers and this results in smaller outputs, this leads to faster training and network efficiency. But as mentioned above there are some drawbacks for pooling layers, so we must be confident when use them.

Exercise 4

A quadratic function which is a function that has a ‘U’ shape when it will plot and make it convex function. This allows us in future when we want to use gradient descent for optimizing the cost function, we won’t get stuck in a local optimum. This function is also called Mean Squared Error.

MSE loss is used in regression tasks where we are trying to minimize an expected value of some function on our training data. MSE is calculated by taking the difference between true ‘y’ and our prediction, then square those values. We take these new numbers (square them), add all of that together to get a final value, finally divide this number by ‘y’ again. This will be our final result.

$$MSE = \frac{1}{n}\Sigma\left(y - \hat{y}\right)^2$$

Cross-entropy loss is used in classification problem. Where we are trying to minimize the probability of a negative class by maximizing an expected value of some function on our training data. It is used to measure the difference between two probabilities that a model assigns to classes. Cross-entropy loss is calculated by taking the difference between our prediction and actual output. Here is the formula for calculating cross entropy loss:

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

Cross entropy will work best when the data is normalized (forced between 0 and 1) as this will represent it as a probability. This normalization property is common in most cost functions.

Exercise 5

ReLU activation function is a type of activation function that is used in neural networks. It is a simple and fast method for implementing nonlinear functions. ReLU Activation Function takes the maximum value of x in each instance. Additionally, the ReLU Activation Function tends to converge faster since it has no division or exponential properties in its calculations. The problem with the use of ReLU is when the gradient has a value of 0. In such cases, the node is considered as a dead node since the old and new values of the weights remain the same. This situation can be avoided by the use of a leaky ReLU function which prevents the gradient from falling to the zero value.

$$f(x) = max(0, x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$

The Leaky ReLU Activation Function is very similar to the ReLU Activation Function with one change. Instead of sending negative values to zero, a very small slope parameter is used which incorporates some information from negative values.

$$f(x) = max(0, x) = \begin{cases} x & x > 0 \\ 0 & mx \leq 0, \text{where } m \text{ is a preselected slope value} \end{cases}$$

As mentioned above, Relu is faster function compared to leaky Relu because it doesn’t have any division or exponential in its calculation.

The vanishing gradient problem arises from the nature of the partial derivative of the activation function used to create the neural network. The problem can be worse in deep neural networks using Sigmoid activation function. To address problem of vanishing gradients, we can use Relu and its variants like LReLU, PReLU, ELU, SELU.

Exercise 6

In this exercise, we are going to implement CNN for CIFAR10 classification. CIFAR10 dataset has 50,000 samples as training set and 10,000 samples as testing set which consists of 10 classes.

- **Data Loading and preprocessing**

  First, we set the seed for reproducibility. We split our dataset into train, validation and test sets. We considered 40,000 examples as train set, 10,000 as validation set and 10,000 as test set.

  We used batch size of 64 and we didn't apply any special transformation, we just converted the images to Tensor.

- **Models**

  1) **Simple CNN**

     For first model, we used two convolutional layers with 8 and 16 filters following by max pooling and a fully connected layer for decision making. We used cross entropy as loss function and Adam optimizer with learning rate of 0.001. We trained the model for 15 epochs. After training our model, we got 64.54% accuracy on train set and 61% accuracy on test set.

  2) **Second model**

     In this model, we used more convolutional layers with batchnorm. We also used global average pooling after last convolutional layer and we after that we use fully connected layer for decision making. Our convolutional layers have 32,64 and 128 filters respectively. We used batchnorm after each convolutional layer. Also, we applied max pooling after each conv layer. We set Relu as hidden layers activation function. After training the model with the same config as first model, we got 81.39% accuracy on train set and 72.95% accuracy on test set. We can see huge improvement on both accuracies.

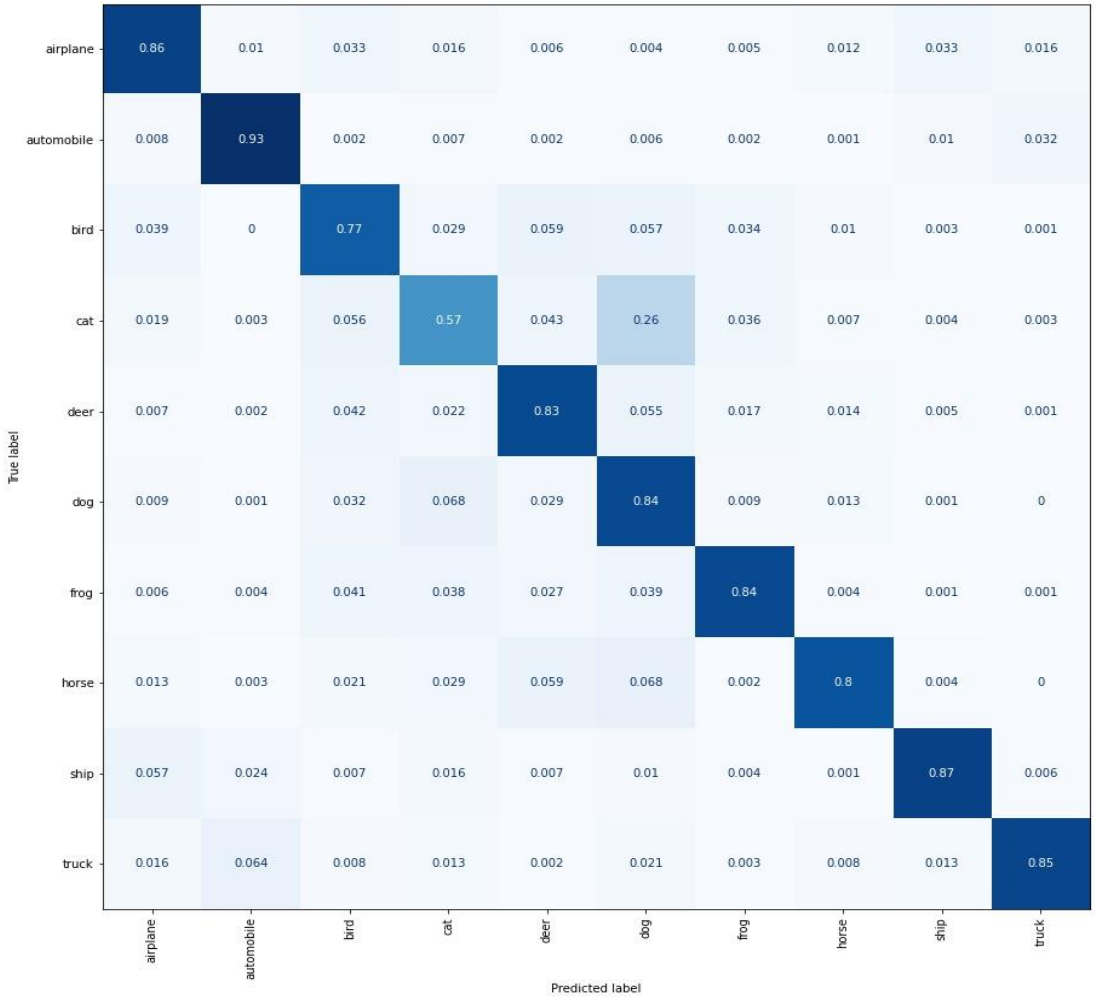
  3) **Third model**

     In third model, we used 8 convolutional layers, two 32 filters conv layers following by max pool, two 64 filters conv layers following by max pool, two 128 filters conv layers following by max pool, two 256 filters conv layers following by max pool. We applied batchnorm after each conv layer. Also, we used leaky Relu as activation function and global average pooling after last conv layer. To prevent overfitting, we used dropout

after global average pooling with probability of 50%. After training the model for 15 epochs, we got 96.88% accuracy on train set and 81.90% accuracy on test set.

- **Confusion Matrix**

  We calculated confusion matrix for last model on test set. In figure below you can see the confusion matrix. We use normalized version of confusion matrix. As you can see, our model has good performance almost on all classes except ‘bird’ and ‘cat’. 57% of samples which belongs to ‘cat’ class predicted true, and 26% of them are predicted as ‘dog’. 77% of ‘bird’ class are predicted well and 6% of them are predicted as ‘deer’ and 5% of them as ‘dog’.



References

- https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/
- https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c

- https://towardsdatascience.com/neural-network-breaking-the-symmetry-e04f963395dd
- https://analyticsindiamag.com/comprehensive-guide-to-different-pooling-layers-in-deep-learning/
- https://medium.com/machine-learning-for-li/a-walk-through-of-cost-functions-4767dff78f7
- https://www.nomidl.com/deep-learning/difference-between-leaky-relu-and-relu-activation-function/
-