EE569: Homework #5

Yifan Wang
wang608@usc.edu #3038184983

March 19, 2019

1 CNN Architecture and Training

1.1 Architecture

1.1.1 Fully Connected Layer

Fully Connected (FC) Layer, as its name implies each neuron connects with all activations from previous layer with different weights. Which can be explained as learning different features' complex combination. It has be proved in mathematics that FC can fit almost every curves no matter how complex it is. Several fully connected layer in the end of a CNN can be regarded as a multi layer perceptron. It serves as a classifier which combines local features extracted by convolution layers into global features and classify them to different class. This means spatial information is lost when passing through FC layers. For recognition problem it would not affect final result. When comes to segmentation or pixel-wise labeling, it does matter so that FC layers are replaced by convolutional and transpose convolutional layers.

FC layer contains most trainable parameters of a CNN. Dropout is now widely used between FC layers to avoid overfitting. And it requires fixed size input image due to dot product used.

1.1.2 Convolutional Layer

Convolutional layer serves as feature extractors. In small dataset good result can be achieved by using several fully connected layer alone, which is named as MLP where each pixel value becomes a feature. However, with a large image, lots of parameters are need which means more storage, heavier computation, longer training time. Besides, lots parameters means it would super hard to train and overfitting tends to occur anywhere. On the other hand, an image has property that pixels are correlated with their neighbours, so that it would be more efficient and better using convolutional layer, which shares weights and has stronger ability to extract features from images.

In earlier work, like AlexNet, they used large convolutional kernel like 11×11. However, recently most networks use several combination of several 3×3 kernel. In mathematics, a 5×5 kernel can be replaces by two 3×3 cores while achieving same targeted size. While more nonlinearity from two 3×3 kernels lead to a better result. Besides small convolutional kernels would have less trainable parameters. For example, one 5×5 kernel has 25 trainable parameters, while two 3×3 kernels have only 18.

Another special 1×1 convolutional layer is now widely used for increasing or decreasing channels (number of filters). It can be regarded as a kind of FC layer while do not care about input size and shared weight. Besides, convolutional layer tends to preserve spatial information which is useful in relocation, pixel-wise labeling or attention.

1.1.3 Max Pooling Layer

Pooling layer serves as reducing spatial size of feature maps. There are several different pooling strategies like max pooling and averaging pooling. For max pooling, it choose the largest response in pooling window (larger response means more discrimination power), while averaging pooling using mean of all responses.

Max pooling can be regard as a high pass filter which keep more texture information, on the other hand, averaging pooling resembles a low pass filter which emphasis on overall features (less textures and details).

For image classification, max pooling is used in most cases. Since major task is to identify objects, more detail textures would help identification and lead to a better result.

Another benefit that pooling layer brought is reduction of trainable parameters. Less parameters means less memory to save them and less time to train a network. Besides, to many trainable parameters need more training data otherwise it tends to be overfitting.

1.1.4 Activation Function

Activation functions would add non-linearity to network and make it much deeper, since both convolutional and fully connected layer are linear operation. Simply cascading them with out an activation function is similar to one single layer and would not give brilliant performance. One another possible explanation (my understanding) to activation functions is that activation function may help to update gradient in a sparser way. As there are millions of parameters in a network, in such high dimension, almost anywhere is a local minima. If update these parameters at the same time, some may update to wrong directions which lead to a worse performance. While compare all activation functions, they all have property that in some part gradient tends to be zero. With no update in some directions, it is more likely to update to true direction. Resulting a more accurate performance. That might prohibit linear activation from performing better especially in large network.

$Sigmoid$ is used in earlier work,

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

However it has several drawbacks. (1) It is not zero-mean, which means all gradients would updated to same direction (bound sign of gradients together with either negative or positive), so that more time is needed before convergence. (2) Gradients of sigmoid when input is large or small would be almost zero, which means no or little update would happen, leads to lots of dead neurons. Once a neuron died, it would have little chance to be active again. While there are generally about 20 percent neurons is died in a network which would not affect performance (in some theories, it would be a good thing since human brain have lots of none activated neurons as well). But too many died neurons would definitely affect performance. (3) Computing sigmoid needs exp which is costly. So that, sigmoid is almost not used, expect using in output layer of a binary classifier.

$tanh$ solves non-zero mean problem, while the other two problems of sigmoid remain. It performs better than sigmoid, especially it can expand distance between two different features. Due to its zero mean character, it can accept a relative large learning rate.

$$a_{tanh}(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

$ReLU$ is a super popular activation function nowadays especially in image fields which is designed to mimic neurons in human brain. It is easy to compute, never saturate in positive axis which is good. However, it is not zero-mean, so that a relative small learning rate is needed otherwise, lots of neurons would be died. Several variants like $LeakyReLU$, ELU is developed to partly ease this problem.

$$a_{ReLU}(x) = max(x, 0)$$

$$a_{Leaky-ReLU}(x) = max(x, 0.001x)$$

$Maxout$ choose the maximum response of in same position of several networks. It performs better without having saturation problem or slow updating speed. But several networks means far more parameters is used and training time would increase dramatically.

$$a_{max-out}(x) = max(w_1x + b_1, w_2x + b2, ...)$$

1.1.5 Softmax Function

$$a_{softmax}(x_i) = \frac{e^{x_i}}{\sum e^{x_i}}$$

Generally, output of a fully connected layer would vary greatly, softmax serves as a normalizer which normalize these response to $[0, 1]$, so that it would be easier to compute loss. Besides, sum of all classes would be 1, so that softmax can be regard as turn response into probability that represents input belongs to this class.

It is a kind of soft decision. While a hard decision like one-hot can be used to choose largest output response as finally predict result. It comes the problem that we cannot decide the degree of right. For example, a three class MLP output $[-1, 2, 4]$, with one-hot ground truth is $[0, 0, 1]$, if using hard decision in this case loss would be zero, no update would happen, which comes the same case when output is $[-1, 2, 40]$. Comparing the loss, they are all zero, but it is a general agreement that later one would have much higher possibility that prediction is true than former one. While using softmax function, $[-1, 2, 4]$ becomes $[0.0059, 0.118, 0.876]$, $[-1, 2, 40]$ becomes $[1e-18, 3e-17, 0.999]$. The loss would be 0.13 in first case while almost zero in second case.

With this difference, each step would has update, according to the confidence of prediction. A bitter local minima can be found.

1.2 Over-fitting

Over-fitting happens when a huge network is feed with few data. Network tends to fit all these data perfectly with high training accuracy, while it would achieve a relative low test accuracy on testing data. This kind of perfectly fit can be explained as over-interpretation. Some characters that do not happen in real data distribution would be invented by model. For example, suppose P_{data} comes from a line, 3 points on this line with some noise is given. As we all know P_{data} would have two parameters. If only one parameter in this model, it would result a line parallel to x-axis, it is under-fitting. Using two parameters in this model, P_{model} would fits P_{data} nicely. While given a three parameter model, P_{model} would generate a parabola that fit all these three noisy point (loss would be zero, accuracy would be one on these three points during training stage), however with new test data used, test accuracy would be much lower.

For much complex P_{data} no one knows exactly how many parameters needed, to avoid over-fitting there are several techniques:

(1) Early stop, this method does not need to modify the model, what we should to is to check training accuracy and validation accuracy each epoch. Once their gap becomes larger, stopping training and using trained parameters from previous epoch.

(2) Regularization, generally there are L_1, L_2 regularization, or use both methods. Which would add some penalty term to loss function. L_1 would make some gradient to zero, while L_2 would never generate a zero gradient. Both method would help to reduce complexity.

(3) Dropout, using a bigger network (suppose originally needs 10 neurons in this layer, dropout rate is 0.5, in this case 20 neurons are need if using dropout), while during training, chose only a small proportion of neurons which selected randomly. It is a kind of using train many different small networks. When testing, using all neurons and discount each neurons' response with dropout rate. This method would perform much better. Last November, a updated version named targeted dropout is published, which dropout neurons with lower response by propose.

(4) Bagging and boosting might can be regard as methods against over-fitting as well. Bagging trained several weak learner (small network each using part of data or features), and use them to vote for final result. It is same with boosting which would change weight for choosing test data for each iteration and give a confidence weight for each model when voting according to their accuracy.

1.3 Why better

Main reason that CNN works better than general method is that it can derive more features from each image and some complex models that are unknown, while previous works have less features and relative

simple models. For example, image-net competition 2012 [2], Alexnet get super performance over other classifiers which use traditional methods like SIFT, Fisher Vectors, color etc. Traditional methods all need carefully human design but no one knows what more representative features can be used or how to combine these features especially when there are millions of data. On the other hand, CNN can adaptively extract features that human can not design or find some rules that is unknown from these huge dataset through training. So that it would give super performance especially on large complex dataset where it would be difficult for human to find some general rules.

1.4 Loss Function and Back Propagation

Most popular loss function for a classification problem is cross entropy.

$$L = -\sum y(i)log(a(i))$$

where y is ground truth label, a is output of softmax function. It can be used to evaluate the similarity of true label with predicted label. More similarity would give less loss. For example binary classification:

$$L = -y(0)log(a(0)) - (1 - a(0))log(1 - a(0))$$

When true label is 0 while prediction $P(0) = 0.5$, loss would be 0.69, if $P(0) = 0.99$ $L = 0.01$. What we should do is to minimize this loss on training data according to this measurement.

But generally optimize loss function would be non-convex, optimizing it would be a NP hard problem, since in such high dimension almost everywhere can be a local minima. Optimizer would stack in such points. In that case SGD is used to find or approach global minima. If y is one-hot label, for one particular data belongs to k^{th} class, loss function can be reduced to:

$$C = -\ln a_k$$

With the help of softmax activation, output of other features is embedded in a_k, so that it would be enough to use a_k represent them all. Then, derivatives of C is:

$$\nabla C = \begin{bmatrix} 0 \\ \dots \\ \frac{\partial C}{\partial a_k} \\ \dots \\ 0 \end{bmatrix}$$

$$\delta = a - y$$

δ is the gradient of output layer, for fully connected layer, it would update like a general MLP, using following formula which is based on chain rule:

$$\delta^l \longleftarrow (a^{l-1})' \odot [(W^{l+1})^T \delta^{l+1}]$$

$$W^l \longleftarrow W^l - \eta \delta^l [a^{l-1}]^T$$

$$b^l \longleftarrow b^l - \eta \delta^l$$

Back propagation through max pooling is somehow similar to using maxout activation. Gradients would only pass through where the maximum response is chosen, others are set to be zero which means location of max response is needed to be memorized during forward process. While average pooling means separate gradients into equal parts and send to all response in this window. Similar pass or no pass of gradient happens on dropout layers, where gradients only pass through neurons that are chosen in this stage.

For convolution layer, it is similar to fully connected layer, since both dot product and convolution are linear operations.

$$\delta^l \longleftarrow (a^{l-1})' \odot [\delta^{l+1} * flip(W^{l+1})]$$

$$W^l \longleftarrow W^l - \eta\delta^l * [a^{l-1}]^T$$
$$b^l \longleftarrow b^l - \eta\delta^l$$

While in above cases, a learning rate η is needed to be specify which may cost lots of time and effort to tune these hyper parameters. And if not control well, SGD would not converge fast or even unstable. Besides, some cases would update gradients to wrong direction or stack into local minima.

In that case, some techniques like momentum is used, which memorized a history gradient update with a discount rate. If present update direction is the same as history gradient, update would be enhanced, else it would be corrected to some direction close to history gradient. It is a kind of low pass filter to smooth gradient update. Nesterov momentum which is derived from a much mathematics way. According to proof, it should converge much faster thought choosing orthogonal direction and most accurate update step, while in reality more computation is need which cost more time. RMSprop where no learning rate is needed, learning rate would be adjust adaptively. Adam combine RMSprop with momentum resulting a generally used optimizer.

Besides, other methods like batch normalization is used to speed up convergence by normalizing input of each layer. Data's mean and other statistics would change after passing each layer When using min-batch, gradient direction tends to vary with change of each min-batch. While batch normalization serves as fixing the data's statistics property, resulting a more smooth update curve, faster converging speed and higher accuracy.

2 Train LeNet-5 on MNIST

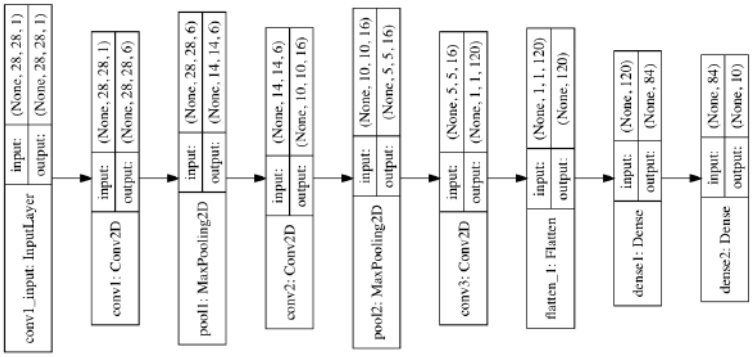
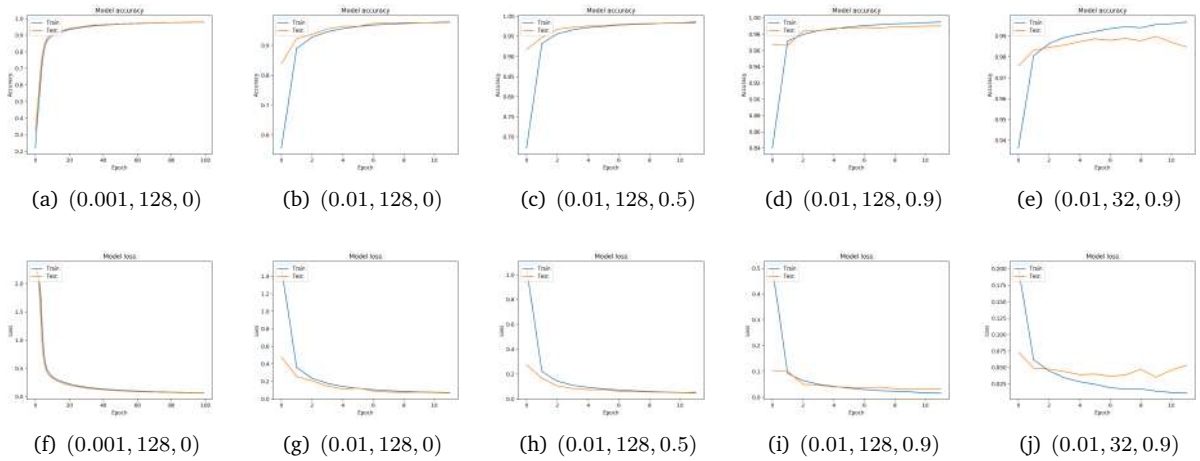


Figure 1: LeNet-5 model

Setting	#1	#2	#3	#4	#5	Overall	variance
$(0.001, 128, 0)$	0.9798	0.9787	0.9789	0.9783	0.9778	0.9787	4.4e-7
$(0.01, 128, 0)$	0.9839	0.9869	0.9843	0.9826	0.9830	0.9841	2.3e-6
$(0.01, 128, 0.5)$	0.9868	0.9880	0.9889	0.9878	0.9828	0.9869	4.6e-6
$(0.01, 128, 0.9)$	0.9905	0.9904	0.9900	0.9883	0.9902	0.9899	6.5e-7
$(0.01, 32, 0.9)$	0.9898	0.9909	0.9892	0.9867	0.9897	0.9893	1.9e-6
$(0.01, 64, 0.9)$	0.9901	0.9878	0.9889	0.9890	0.9927	0.9897	2.8e-6

Table 1: test accuracy on LeNet-5 using SGD with different settings
$(lr, minbatch, momentum)$ no learning rate decay

For different parameters, it can be found from $Figure2$ and $Table1$, where optimizer uses SGD. A smaller learning rate $Figure2(a, b)$ would converge much slower ($(a), epoch = 100$, $(b), epoch = 12$) and it is likely to be stuck in some local minima, since each update is pretty small. On the other hand, if choosing a learning rate much larger than 0.01 would make update unstable, a less smooth learning curve would occur, overflow or model with lots of died neurons might happen when learning rate is too large. While considering using



(a) $(0.001, 128, 0)$ (b) $(0.01, 128, 0)$ (c) $(0.01, 128, 0.5)$ (d) $(0.01, 128, 0.9)$ (e) $(0.01, 32, 0.9)$

(f) $(0.001, 128, 0)$ (g) $(0.01, 128, 0)$ (h) $(0.01, 128, 0.5)$ (i) $(0.01, 128, 0.9)$ (j) $(0.01, 32, 0.9)$

Figure 2: parameter test using SGD in Keras
$(lr, minbatch, momentum)$ no learning rate decay

momentum ($Figure2(b, c, d)$, larger momentum would result faster convergency. Until size of min-batch, small min-batch would update more noisy (since some update may offset others or much random direction, larger variance on test accuracy. Final accuracy would be almost the same.) and would not use computational power efficiently, while if using full batch update might not possible on some large dataset, generally a min-batch size of $128 - 1024$ would perform well (depends on computer power according to roof theory, small min-batch take more time to train). Learning rate decay would help to approximate minima more accurate. A relatively large learning rate can help to converge faster at beginning of training, when it is nearly converge, a small learning rate can help to find best point, and resulting a more accurate training and test result.

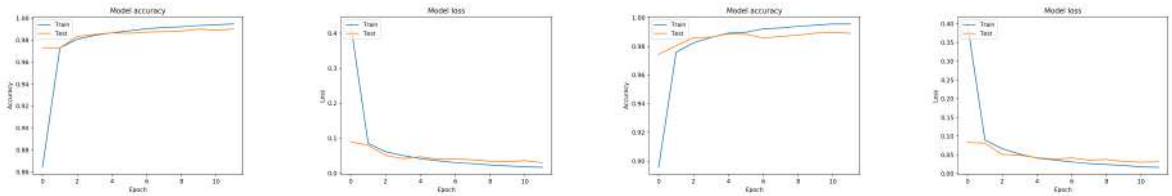
For different optimizer, when all use default setting as Keras provided. it can be found that compare with SGD, Adam and RMSprop would converge much faster and much smoother (using SGD+momentum can speed up as well), which means that they required less time to get the same training effect. This is a result from using momentum and adaptive learning rate. What's more, using SGD requires carefully choose learning rate which might require lots of experience and intuition (controls well might perform even better than Adam).

For initialization methods, a small network like this is enough using random number from unit uniform distribution or normal Gaussian with a scaler 0.01. But it is not acceptable if initialize them into same number, otherwise, this network can be reduce to a network only have one single neuron. This is due to that all same weight means all gradient would update to same direction with same magnitude. While for some large deep network, some special initialization methods are needed to avoid vanishing gradient problem. Due to the relative small magnitude of weight, when feed backward to front layer, gradient would almost be zero when pass through several layers. Given a near zero gradient which means almost no update would happen, these weight are not trained at all. In that case, how to initialize a super deep network is a major research filed in CNN. Different methods like $he - normal$, $lecun - normal$, $Xavier - initalization$ are proposed.

Trial	$\#1$	$\#2$	$\#3$	$\#4$	$\#5$	Overall	variance
Test Accuracy	0.9902	0.9899	0.9902	0.9888	0.9904	0.9899	3.3e-7
Test Loss	0.0301	0.0308	0.0292	0.0373	0.0312	0.0317	8.2e-5

Table 2: LeNet-5 using $SGD(lr = 0.01, momentum = 0.9, decay = 0.0001, nesterov = False)$, $minbatch = 128$

The best training result comes from using SGD with $lr = 0.01$, $momentum = 0.9$, $decay = 0.0001$. Five training trials using this setting is shown in $Table2$ and part of learning cures is shown in $Figure3$.



(a) #1 accuracy (b) #1 loss (c) #2 accuracy (d) #2 loss

Figure 3: part of results using $SGD(lr = 0.01, momentum = 0.9, decay = 0.0001, nesterov = False)$

Trial	#1	#2	#3	#4	#5	Overall	variance
Test Accuracy	0.2448	0.2992	0.3779	0.3825	0.3428	0.3294	0.0027
Test Loss	2.6101	2.5533	2.8830	2.3499	2.5404	2.5873	0.0300

Table 3: LeNet-5 trained by positive, tested by negative data

2.1 Negative Images

When applying negative image to trained model in previous section which used only positive images as training data. Test accuracy is shown in $Table3$. It can be found that same test image with only difference in positive or negative, test accuracy drop dramatically. This can be explained that CNNs are actually data based model, they can learn and fit the distribution of data P_{data} with P_{model} nicely. When using positive images, both train data and test data all comes from same distribution which means P_{train} and $P_{test_{pos}}$ is the same as P_{data}. However, when changing test image from positive to negative, $P_{test_{pos}}$ becomes $P_{test_{neg}}$. While $P_{test_{neg}}$ would not agree with P_{train} (linear transformation is needed to make them the same) since they do not come from the same model. That's the reason why CNN would not perform well. It can be understood that CNN had never seem such images.

2.1.1 LeNet-5 trained by both positive and negative images

Trial		#1	#2	#3	#4	#5	Overall	variance
Test Accuracy	positive data	0.9902	0.9893	0.9913	0.9885	0.9905	0.9899	9.5e-7
	negative data	0.9904	0.9899	0.9910	0.9904	0.9899	0.9903	1.7e-7
Test Loss	positive data	0.0323	0.0350	0.0298	0.0370	0.0306	0.0329	7.3e-6
	negative data	0.0290	0.0351	0.0311	0.0323	0.0361	0.0327	8.0e-6

Table 4: LeNet-5 trained by both positive and negative data

One simply solution to this problem is provide network with both positive and negative images as training data. So that when input either negative or positive images it can perform equally well. Results are shown in $Table4$. After trained by both images, it can be conclude that this network has seem negative images. So that when using negative images for testing, it knows how to deal with them and give a good performance.

2.1.2 Preprocess Data

It can be found that positive and negative images's difference are linear ($I_{pos} = 255 - I_{neg}$). In that case, if preprocess is done on originally positive images, no more negative image is needed in training stage. Preprocess for one positive image I is $I_{processed} = abs(I - mean(I))$ ($Figure4$). Similar preprocess is done on test images before testing stage. Through using this processing method, both positive and negative images

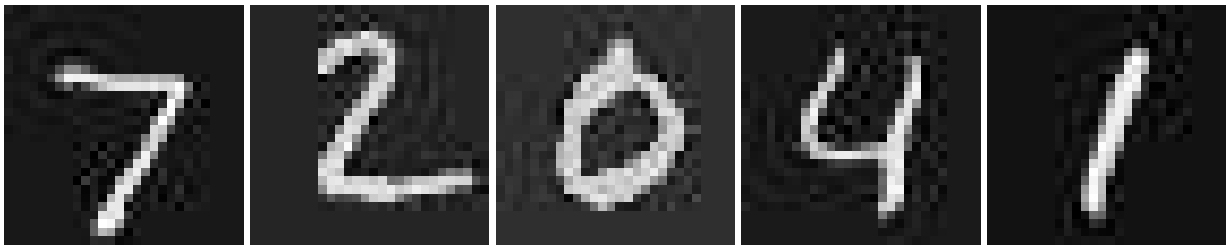


Figure 4: visualization of preprocessing

Trial		#1	#2	#3	#4	#5	Overall	variance
Test Accuracy	positive data	0.9900	0.9872	0.9889	0.9880	0.9893	0.9887	9.7e-7
	negative data	0.9900	0.9872	0.9889	0.9880	0.9893	0.9887	9.7e-7
Test Loss	positive data	0.0322	0.0370	0.0342	0.0339	0.0322	0.0339	3.1e-6
	negative data	0.0322	0.0370	0.0342	0.0339	0.0322	0.0339	3.1e-6

Table 5: LeNet-5 with preprocessed data

would becomes the same if they are originate from same image. That is the reason why test accuracy ($Table5$) is alway the same in one single test for both positive and negative data.

References

[1] `http://cs231n.stanford.edu`

[2] `http://image-net.org/challenges/LSVRC/2012/results.html#t2`