University of Illinois Fall 2018

ECE 544NA: Pattern Recognition
Lecture 10: November 27

Lecturer: Alexander Schwing Scribe: Tianyu Chen

1 Introduction

This lecture talks about deep nets, or in other words, neural networks. We will be discussing the general form for deep nets, the computation graph, the most common layers of deep nets, the training function for deep nets and some mostly used loss functions in deep nets.

2 Why Deep Nets

Last lecture we have discussed multiclass classification, and we came up with this general framework.

$$\min_{\boldsymbol{w}} \frac{C}{2}\|\boldsymbol{w}\|_2^2 + \sum_{i\in\mathcal{D}} \epsilon \ln \sum_{\hat{y}} \exp \frac{L(y^{(i)},\hat{y}) + \boldsymbol{w}^T\psi(x^{(i)},\hat{y}))}{\epsilon} - \boldsymbol{w}^T\psi(x^{(i)},y^{(i)})$$

Figure 1: Our current solution

Limitation : $\mathbf{w^T}\psi(x^{(i)},\hat{y})$ is a linear function between $\mathbf{w^T}$ and $\psi(x^{(i)},\hat{y})$. $\mathbf{w^T}$ is the parameter that we optimize during training process. However, we do not always want to do dot product between our weight vector $\mathbf{w^T}$ and our feature data $\psi(x^{(i)},\hat{y})$. So the solution is to the replace $\mathbf{w^T}\psi(x,y)$ with $\mathbf{F}(\mathbf{w},x^{(i)},\hat{y})$. Both of them should return a real number.

$$\min_{\boldsymbol{w}} \frac{C}{2}\|\boldsymbol{w}\|_2^2 + \sum_{i\in\mathcal{D}} \epsilon \ln \sum_{\hat{y}} \exp \frac{L(y^{(i)},\hat{y}) + F(\boldsymbol{w},x^{(i)},\hat{y})}{\epsilon} - F(\boldsymbol{w},x^{(i)},y^{(i)})$$

Figure 2: More powerful deep nets framework

Next Step: What function $F(w,x,y)$ should we choose for deep nets?

3 Computation Graph

The answer to the last question is that any differentiable composite function will work. So what is the differentiable composite functions? Indeed, it is simply a sequence of functions that operates on the results of some others. The reasons we want it to be in this form is for us later to be able to perform backpropagation when we want to compute the gradient of this composite function, which will be taught in next lecture.

Composite function: $\mathbf{F}(w,x,y) = f_1(w_1,y,f_2(w_2,f_3(...f_n(w_n,x)...)))$

Question: how do we represent this big bulk of functions in computer ?

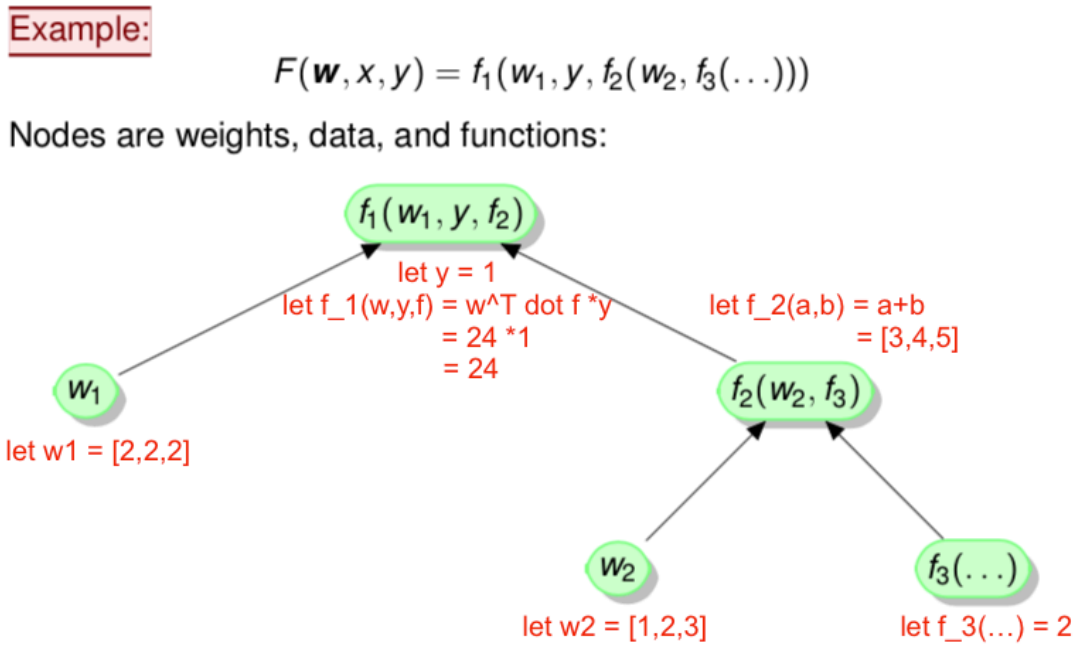

Figure 3: computation graph(acyclic graph)

Explanation: It is the way to formalize the computation process. Indeed, it is just a sequence of computations(functions). For example, when you type in your calculator: 5*4+3. This is also a composite function where you first performed a multiplication(call it f_2) between 5 and 4 , then an addition between the output of f_2 and 3. The node serving input could be arbitrary dimension. It could be a vector, matrix or simply a real number. We have some other nodes that serves as the computation node. With this graph, we will be able to calculate the gradient of arbitrary composite function by performing backpropagation. I have given a really simple example of a sequence of computation in red. The computation starts from bottom. The output spits out at the top node. Note that the computation graph is also the internal representation used by deep nets packages.

Question: What are the individual functions/layers f_1, f_2 in deep nets?

1. Fully connected layers
2. Convolutions
3. Rectified linear units (ReLU): $max0, x$
4. Maximum-/Average pooling
5. Soft-max layer
6. Dropout

We will discuss them one by one.

4 Deep Nets layers

4.1 Fully connected layers

Here's an example of a fully connected layers. As we can see, the input layer and the output layer have the same dimension: 1×5.

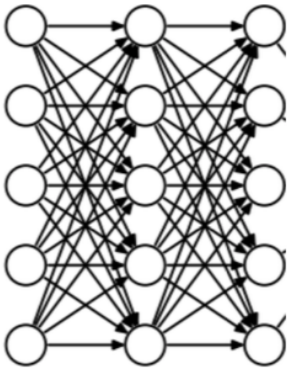

Figure 4: fully connected layers

Question: What dimension should our weight vectors **w** be?

Inorder for the input and output dimension be the same size. The matrix **w** has to be the dimension 5×5. Because if matrix A is of dimension 1×5, only if matrix B is of dimension 5×5 could the product of the AB will be dimension 1×5.

If this example is not clear, let's look at another example, this example is from standford cs231n:

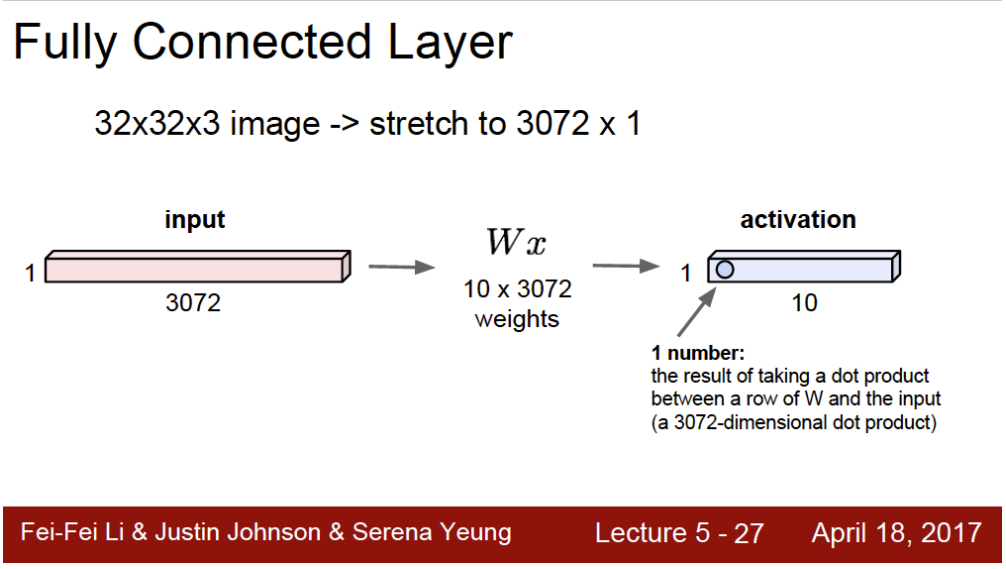

Figure 5: fully connected layers 2

Explanation: The original input image is $32 \times 32 \times 3$. To facilitate our computation, we stretch the image into a long 1D vector. Currently the output vector is only 1×10, different from the input size. This is because it is already the final layer and we would like output the score for each class and no more computation needed. But if we are going to use fully connected throughout the computation, we would like to preserve the image's size, which means to have the output vector output the same dimension.

Question: What's an issue with fully connected layers?

Suppose the input is an image of size 256×256 and if we want the output layer has the identical size. How many weights are necessary?

Solution: Stretch the image into one 1D vector: $256 \times 256 \Rightarrow 1 \times 2^{16}$. If we want the output layer has the idential size 1×2^{16}, as proven above, the weight vector needs to be $2^{16} * 2^{16} = 2^{32}$.

Problem: This computation requires large amount of memory and becomes extremely time consuming. Also the original input dimension is lost on the way since we stretched the image.

4.2 Convolutions Layers

Here's is an example of a convolution layer.

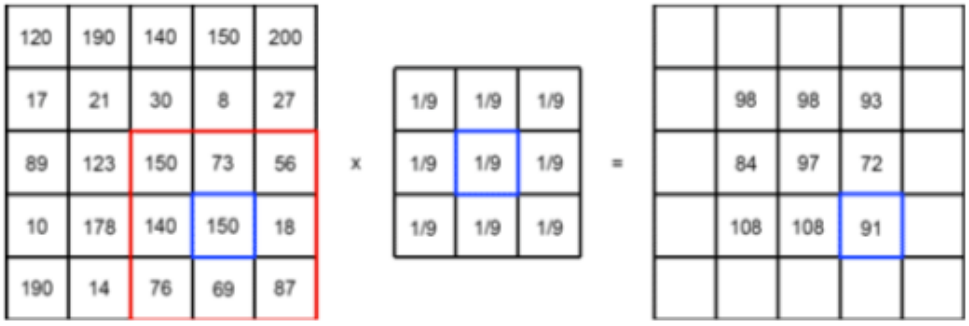

Figure 6: convolution layers

Explanation: Below is a list of term you should know for convolution layers

1. input layer: The square on the left.
2. output layer: The square on the right
3. filter: The square filled with 1/9 in the middle is called a filter.
4. stride: The steps(unit) each time the filter move.
5. filter's width and height: We normally choose the width and height to be the same.

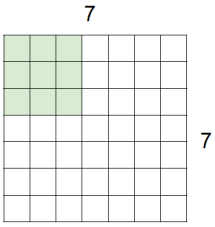

Figure 7: filter with stride 1 of step 1

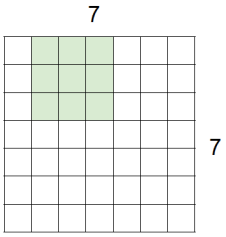

Figure 8: filter with stride 1 of step 2

Example: Above is an example with stride = 1:

Example: Here is another example with stride = 2:

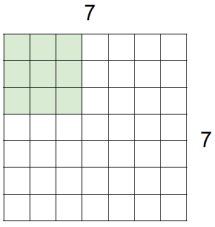

Figure 9: filter with stride 2 of step 1

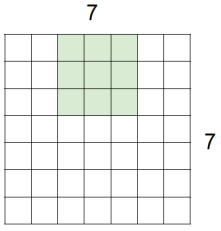

Figure 10: filter with stride 2 of step 2

Currently this output is computed by summing up all the products of input and filter. For example, $150*1/9+73*1/9+56*1/9+140*1/9+150*1/9+18*1/9+76*1/9+69*1/9+87*1/9 = 91$.

4.3 Rectified linear units (ReLU)

Relu is basically a layer that takes the filters out the negative numbers.

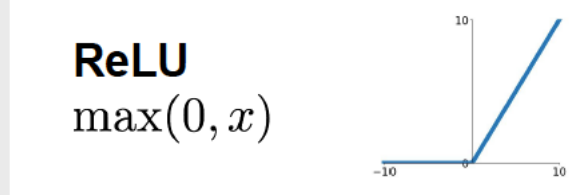

Figure 11: ReLU

4.4 Maximum-/Average pooling

The objective of pooling is to reduce the size of current layer and make it more manageable.

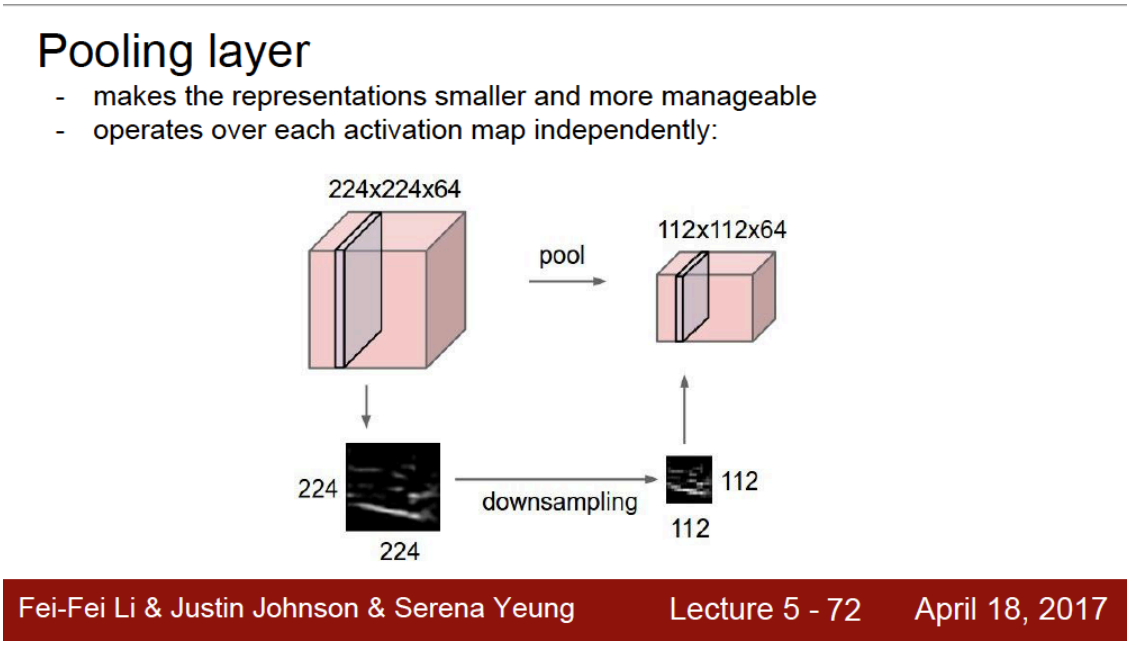

Figure 12: pooling example

Explanation: In the example above, we see that the size of the input matrix was reduced from $224 \times 224 \times 64$ to $112 \times 112 \times 64$. Noticed that the last dimention, which is often channel size is preserved. Pooling also make use of filters.

1. For max pooling, we take the maximum number mapped by that filter.
2. For average pooling, we take the average of all the numbers mapped by that filter.

Here is a max pooling example with 2×2 filters and stride 2, meaning we move 2 units each time. The filters' value doesn't matter here.

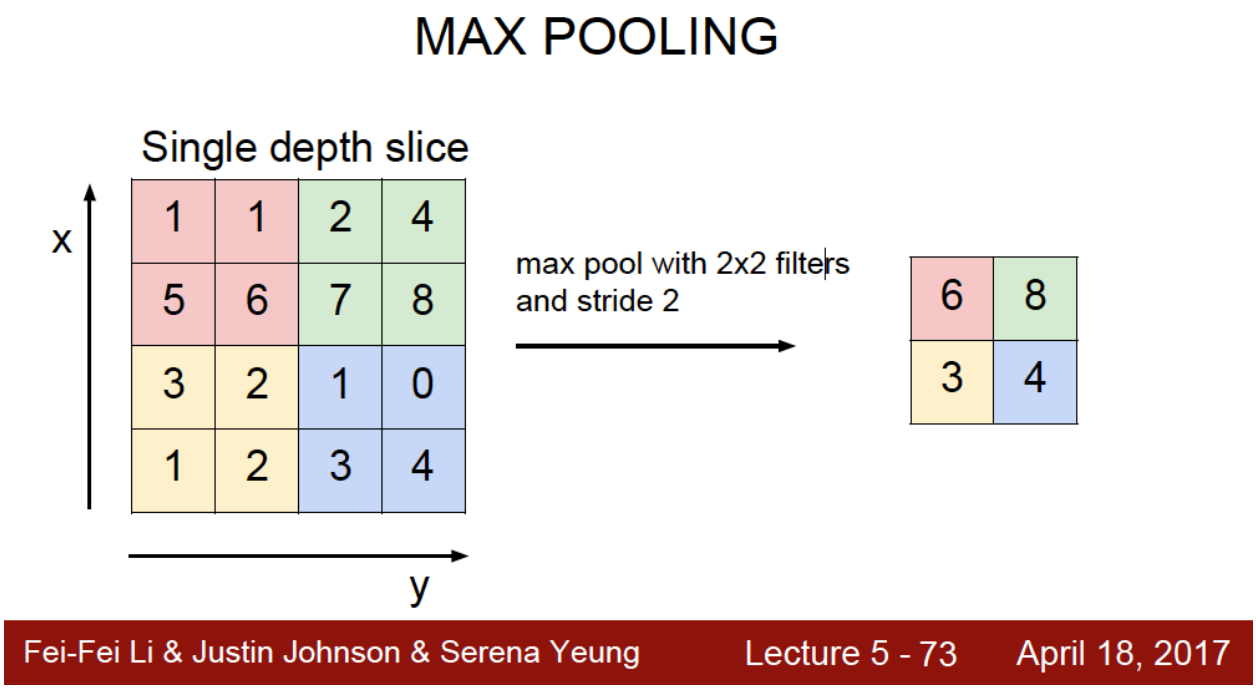

Figure 13: max pooling example

4.5 Soft-max layer

I know I have used a lot of stanford's examples, but please bear with me because these examples make much more sense.

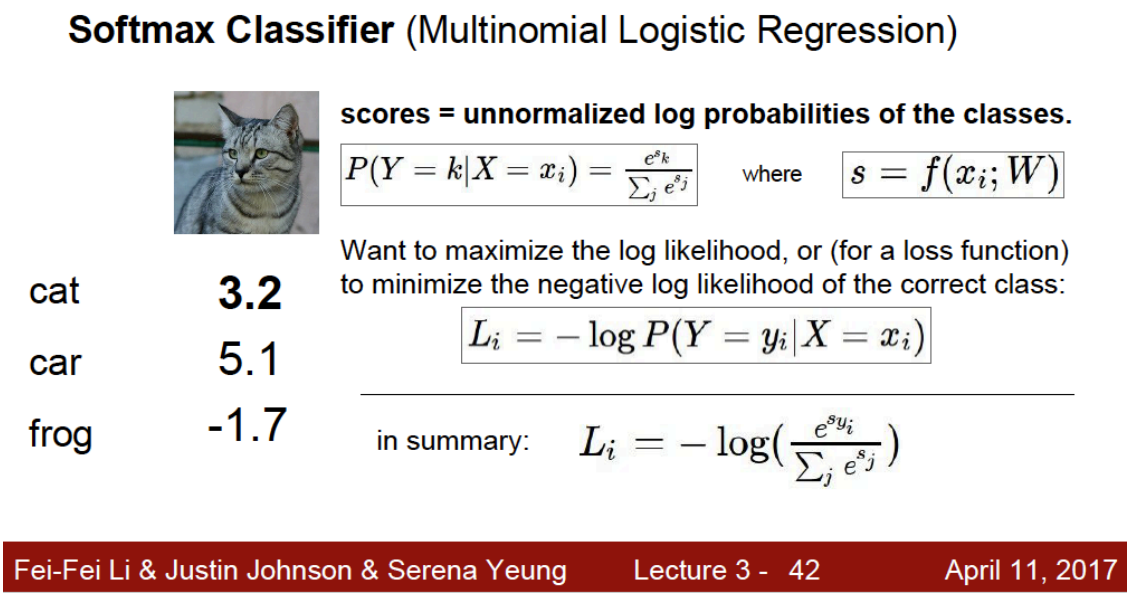

Figure 14: softmax example

Explanation: In the image above, the three number $3.2, 5.1, -1.7$ are the score it received for the

three classes: cat, car, and frog. Apprently, this classification result is not correct because we have incorrectly labeled this image with car.

Soft-max Function:

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \tag{1}$$

Explanation e^{s_k} is the exp of the score k. This function returns the normalized probability of a given class.

As above figure says: we would like to maximize the likelihood, or minimize the negative log likelihood of the correct class:

$$\begin{aligned} L_i &= -logP(Y = y_i|X = x_i) \\ &= -log\frac{e^{s_k}}{\sum_j e^{s_j}} \end{aligned}$$

Here is an example of how we compute the softmax layers.

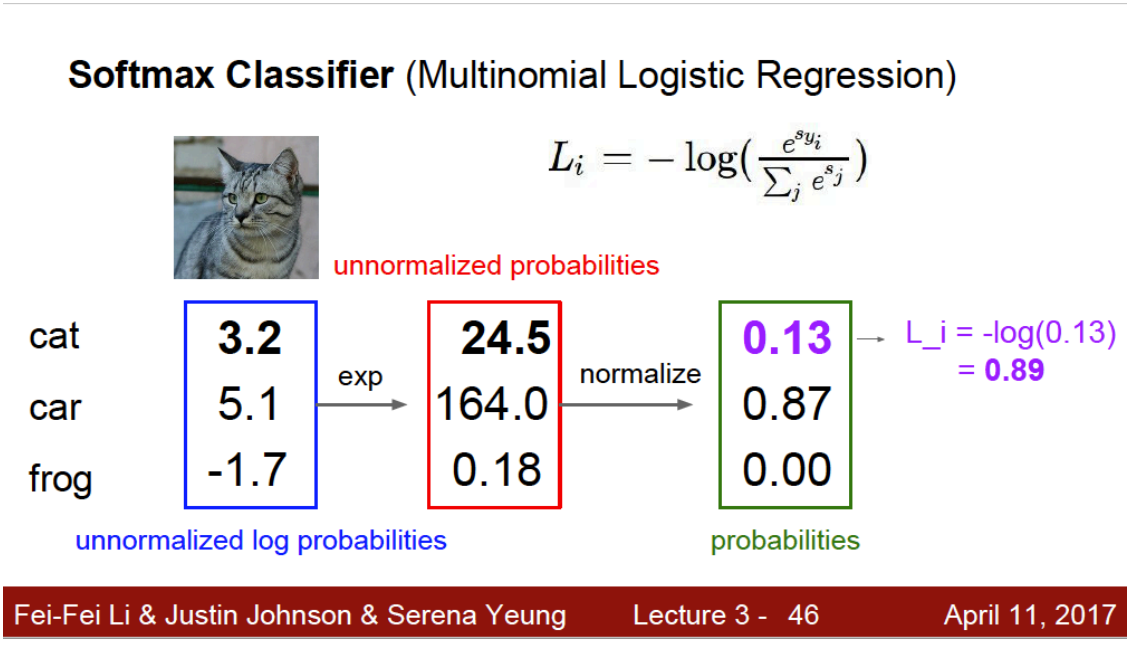

Figure 15: softmax example

Trainable parameter **w**: None is basically meaning there is nothing to train. The softmax layer always give the same output for certain inputs. Because it's merely a function that takes the scores and output the likelihood for one class.

4.6 Dropout Layer

Dropout layers ramdomly sets activations to zero. The purpose of this layer is to prevent overfitting.

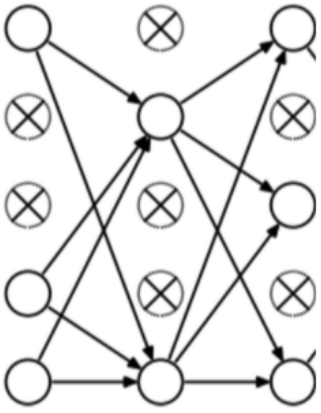

Figure 16: softmax example

5 Example Architecture

After we have introduced all these layers, let's take a look of two industry examples of actuall implementation of these layers.

5.1 LeNet

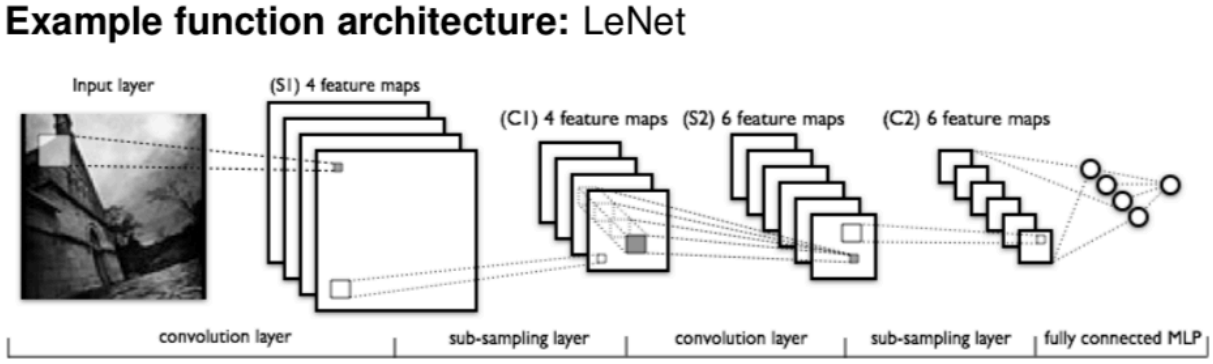

Figure 17: lenet

Explanation: We could see from the graph that the first layer is a convolution layer, the second layer is a sub-sampling layer, then a convolution layer again, a sub-sampling layer, then finally a fully connected layer that output the class.

One thing to notice is that the channels(the number of maps) increase and the resolution of the layer(size of the map) become smaller and smaller as the layers pass the features.

5.2 AlexNet

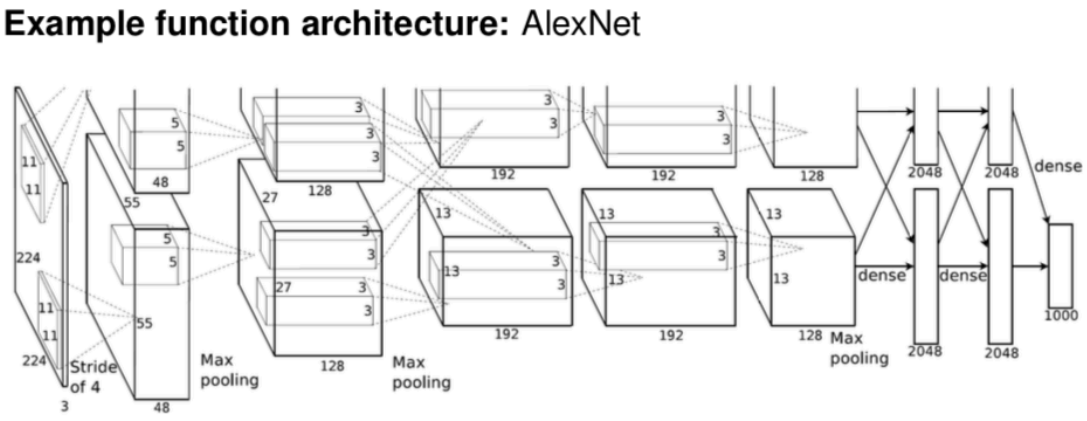

Figure 18: lenet

Explanation: AlexNet contains multiple convolutions layers and also max pooling layers, etc. Different from LeNet, channels in AlexNet is represented by the depth of the cuboid. We can see it increases from $3 \to 48 \to 128 \to 192$ and finally it condense the features into one 1×1000 vector. The reason that the final output is 1000 dimensional is because there are 1000 classes, and we need to have scores for every single classes.

6 Deep Net Training

From the general function, by setting ϵ to 1 and taskLoss to 0, we have the following cost function.

$$\min_{\mathbf{w}} \frac{C}{2} \|\mathbf{w}\|_2^2 + \sum_{i \in D} \ln \sum_{\hat{y}} \exp F(\mathbf{w}, x^{(i)}, \hat{y}) - F(\mathbf{w}, x^{(i)}, y^{(i)}) \tag{2}$$

Now let's introduce a new loss function: cross entropy loss: $-\sum_{i \in D} \sum_{\hat{y}} p_{GT}^{(i)}(\hat{y}) \ln p(\hat{y}|x^{(i)})$. It is a combination of negative log likelihood loss and soft-max function. We will see where the soft-max function is being used in the later derivation. Now the question comes: why minimizing the cost function above is the same as maxing the regularied cross entropy loss function below?

$$\max_{\mathbf{w}} -\frac{C}{2} \|\mathbf{w}\|_2^2 + \sum_{i \in D} \sum_{\hat{y}} p_{GT}^{(i)}(\hat{y}) \ln p(\hat{y}|x^{(i)}) \tag{3}$$

The slides have given us some definitions, but it may still be unclear. That's derive it step by step. First let's flip the max and min.

$$\max_{\mathbf{w}} -\frac{C}{2} \|\mathbf{w}\|_2^2 + \sum_{i \in D} \sum_{\hat{y}} p_{GT}^{(i)}(\hat{y}) \ln p(\hat{y}|x^{(i)}) = \min_{\mathbf{w}} \frac{C}{2} \|\mathbf{w}\|_2^2 - \sum_{i \in D} \sum_{\hat{y}} p_{GT}^{(i)}(\hat{y}) \ln p(\hat{y}|x^{(i)}) \tag{4}$$

Now the left size already looks the same as before. And let's start our derivation on the right side.

$$
\begin{aligned}
-\sum_{i\in D}\sum_{\hat{y}} p_{GT}^{(i)}(\hat{y})\ln p(\hat{y}|x^{(i)}) &= -\sum_{i\in D}\sum_{\hat{y}} \delta(\hat{y}=y^{(i)})\ln p(\hat{y}|x^{(i)}) && \text{plug in the first condition}\\
&= -\sum_{i\in D}\ln p(y^{(i)}|x^{(i)}) && \text{throw away all the zero terms}\\
&= -\sum_{i\in D}\ln\frac{\exp F(\mathbf{w},x^{(i)},y^{(i)})}{\sum_{\hat{y}}\exp F(\mathbf{w},x^{(i)},\hat{y})} && \text{soft-max function}\\
&= \sum_{i\in D}\ln\frac{\sum_{\hat{y}}\exp F(\mathbf{w},x^{(i)},\hat{y})}{\exp F(\mathbf{w},x^{(i)},y^{(i)})} && \text{take care negative sign}\\
&= \sum_{i\in D}\ln\sum_{\hat{y}}\exp F(\mathbf{w},x^{(i)},\hat{y}) && \text{expanded}\\
&\quad -\sum_{i\in D}\ln\exp F(\mathbf{w},x^{(i)},y^{(i)})\\
&= \sum_{i\in D}\ln\sum_{\hat{y}}\exp F(\mathbf{w},x^{(i)},\hat{y}) - F(\mathbf{w},x^{(i)},y^{(i)}) && \text{combine exp and ln}
\end{aligned}
$$

7 Other Loss Functions

Besides regularized cross entropy loss, there are many more loss function we could choose from.

Many more loss functions:

- CrossEntropyLoss

```
loss(x, class) = -log(exp(x[class]) / (\sum_j exp(x[j])))
               = -x[class] + log(\sum_j exp(x[j]))
```

- NLLLoss negative log likelihood

```
loss(x, class) = -x[class]
```

- MSELoss

```
loss(x, y) = 1/n \sum_i |x_i - y_i|^2
```

- BCELoss Binary Cross Entropy: for binary classification

```
loss(o,t)=-1/n \sum_i i(t[i]*log(o[i])+(1-t[i])*log(1-o[i]))
```

- BCEWithLogitsLoss

```
loss(o,t)=-1/n\sum_i(t[i]*log(sigmoid(o[i]))
                 +(1-t[i])*log(1-sigmoid(o[i])))
```

- L1Loss
- KLDivLoss

Figure 19: loss functions

Explanation: the function **loss(x,class)** takes in two parameter **x** and **class**. They represent different things for different functions. For example, for CrossEntropyLoss, **x** is an array contains the scores for each class and **class** is just a index represent the ground truth class. However, for NLLLoss, **x** is an array contains log probabilities for each class. You should always check the parameter in pytorch's documentation before actually plug in any values.

Reference [1].
Reference [2].

References

[1] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[2] F. F. Li., J. Johnson., and S. Yeung. Lectures notes in cs231n, April.