# Homework 5
# 601.482/682 Deep Learning
# Fall 2019

Jason Zhang jzhan127

October 11, 2019

**Due Fri. 10/11 11:59pm.**
**Please submit a latex generated PDF**
**to Gradescope with entry code MKDPGK**

1. Computing Network Sizes

   (a) Given the multi-layer perceptron below in Figure 1, compute the number of free parameters for this model (Note that biases are explicitly disregarded in this case).

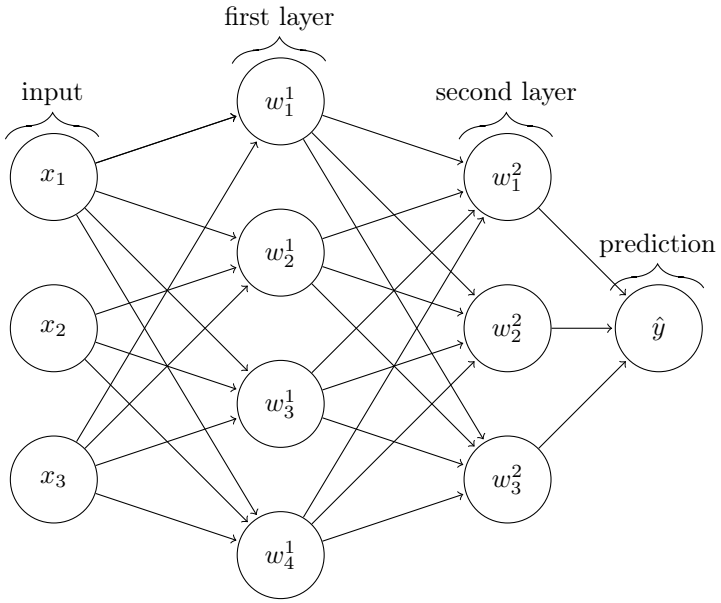



Figure 1: Multi-layer Perceptron

**Answer:**
Calculating free parameters from input layer -> first layer:
x1 is connected to $w_1^1$, $w_2^1$, $w_3^1$, $w_4^1$
x2 is connected to $w_2^1$, $w_3^1$, $w_4^1$
x3 is connected to $w_1^1$, $w_2^1$, $w_3^1$, $w_4^1$.
Summing those connections we have a total of 11 free parameters in this layer.
Calculating free parameters from first layer -> second layer:
$w_1^1$ is connected to $w_1^2$, $w_2^2$, $w_3^2$
$w_2^1$ is connected to $w_1^2$, $w_2^2$, $w_3^2$
$w_3^1$ is connected to $w_1^2$, $w_2^2$, $w_3^2$
$w_4^1$ is connected to $w_1^2$, $w_2^2$, $w_3^2$
Summing these connections we have a total of 12 free parameters in this layer.
Calculating free parameters from the second layer -> prediction:
$w_1^2$ is connected to $\hat{y}$
$w_2^2$ is connected to $\hat{y}$

$w_3^2$ is connected to $\hat{y}$
Summing these connections we have a total of 3 free parameters in this layer.
Summing all of the free parameters in this network we get:
11 + 12 + 3 = **26 Free Parameters**.

In this part, you will analyze a state-of-the-art architecture not discussed in class. In the ImageNet ILSVRC 2014 contest, two fairly deep networks performed very well: The VGG network and the GoogLeNet. While VGG performed best in the localization task and ranked second in classification, GoogLeNet won the classification task achieving a top-5 error rate of 6.67%. Figure 2 presents the architecture of GoogLeNet, which is built up of 9 stacked "inception" modules displayed in Figure 3.

(b) Consider the layer "inception (3a)" from Table 1(Figure 4 ) in the GoogLeNet paper. Notice how "3x3 reduce" and "5x5 reduce" are used between layers - from section 5, this "stands for the number of 1x1 filters in the reduction layer used before the 3×3 and 5×5 convolutions". Compute the number of free parameters for the "inception (3a)" layer. Show the detailed steps of how you compute the number. Note that in the 'params' column of Figure 4, 1K=1024. Also, GoogleNet uses no bias in the Inception module.
**Answer:**
The input to inception (3a) is 28x28x192. Also assuming there's no bias in the inception module(as per instructions).
Calculating free parameters after the 1x1 Convolutions we get:
(1x1x192)x64, where 1x1x192 corresponds to the convolution kernel and 64 is the resulting outputted feature maps (using this logic throughout the calculation).
Thus we get (1x1x192)x64 = 12288.
Calculating free parameters after 3x3 reduction we get:
(1x1x192)x96 = 18432
Calculating free parameters after 3x3 convolution we get:
(3x3x96)x128 = 110592
Calculating free parameters after 5x5 reduction we get:
(1x1x192)x16 = 3072
Calculating free parameters after 5x5 convolution we get:
(5x5x16)x32 = 12800
Calculating free parameters after pool proj layer:
(The 1x1 convolution after the max pool gives us extra parameters): (1x1x192)x32 = 6144
In total we have 12288 + 18432 + 110592 + 3072 + 12800 + 6144 = **163,328 free parameters**.

(c) Now, consider that the reduction portion of "3x3 reduce" and "5x5 reduce" were omitted (i.e. no 1x1 filters were used before the 3x3 and 5x5 convolutions). Compute the number of free parameters for this updated "inception (3a)" layer. How does this compare to the original layer?
**Answer:**
Removing the 3x3 reduce and 5x5 reduce we compute the following: Free parameters after 1x1 Convolution: (1x1x192)x64 = 12288
Calculating free parameters after 3x3 convolution we get:
(3x3x192)x128 = 221184
Calculating free parameters after 5x5 convolution we get:
(5x5x192)x32 = 153600
No more 1x1 convolutions after the max pool layer since we're referring to the naive version.
In total we have 12288 + 221184 + 153600 = **387072 free parameters**.
It is significantly more parameters than the non naive version which is more memory required. The reduction is more computationally efficient and thus more feasible.

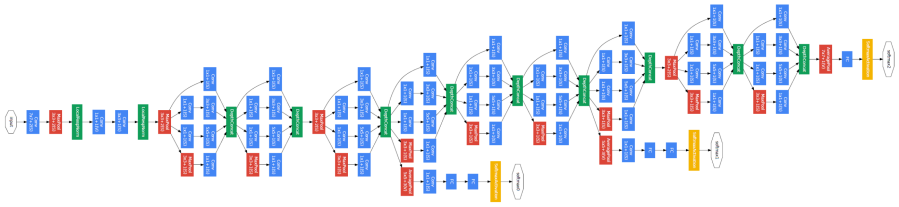
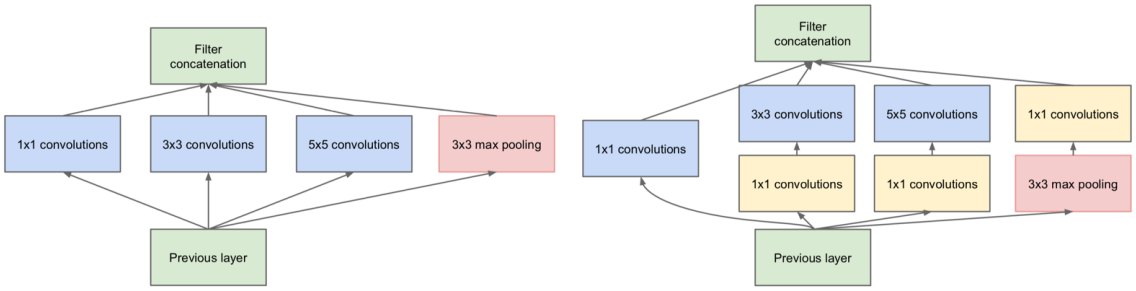


Figure 2: GoogleNet Architecture




(a) Inception module, naïve version

(b) Inception module with dimensionality reduction

Figure 3: Inception Module

| type | patch size/ stride | output size | depth | #1×1 | #3×3 reduce | #3×3 | #5×5 reduce | #5×5 | pool proj | params | ops |
|---|---|---|---|---|---|---|---|---|---|---|---|
| convolution | 7×7/2 | 112×112×64 | 1 | | | | | | | 2.7K | 34M |
| max pool | 3×3/2 | 56×56×64 | 0 | | | | | | | | |
| convolution | 3×3/1 | 56×56×192 | 2 | | 64 | 192 | | | | 112K | 360M |
| max pool | 3×3/2 | 28×28×192 | 0 | | | | | | | | |
| inception (3a) | | 28×28×256 | 2 | 64 | 96 | 128 | 16 | 32 | 32 | 159K | 128M |
| inception (3b) | | 28×28×480 | 2 | 128 | 128 | 192 | 32 | 96 | 64 | 380K | 304M |
| max pool | 3×3/2 | 14×14×480 | 0 | | | | | | | | |
| inception (4a) | | 14×14×512 | 2 | 192 | 96 | 208 | 16 | 48 | 64 | 364K | 73M |
| inception (4b) | | 14×14×512 | 2 | 160 | 112 | 224 | 24 | 64 | 64 | 437K | 88M |
| inception (4c) | | 14×14×512 | 2 | 128 | 128 | 256 | 24 | 64 | 64 | 463K | 100M |
| inception (4d) | | 14×14×528 | 2 | 112 | 144 | 288 | 32 | 64 | 64 | 580K | 119M |
| inception (4e) | | 14×14×832 | 2 | 256 | 160 | 320 | 32 | 128 | 128 | 840K | 170M |
| max pool | 3×3/2 | 7×7×832 | 0 | | | | | | | | |
| inception (5a) | | 7×7×832 | 2 | 256 | 160 | 320 | 32 | 128 | 128 | 1072K | 54M |
| inception (5b) | | 7×7×1024 | 2 | 384 | 192 | 384 | 48 | 128 | 128 | 1388K | 71M |
| avg pool | 7×7/1 | 1×1×1024 | 0 | | | | | | | | |
| dropout (40%) | | 1×1×1024 | 0 | | | | | | | | |
| linear | | 1×1×1000 | 1 | | | | | | | 1000K | 1M |
| softmax | | 1×1×1000 | 0 | | | | | | | | |

Figure 4: GoogLeNet incarnation of the Inception architecture.

2. Receptive Fields

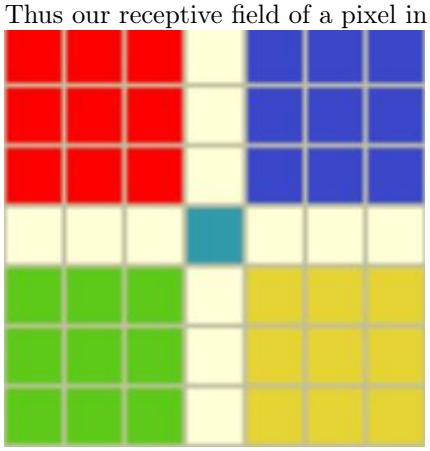
(a) Consider the network architecture in Figure 5 (You can ignore the information of $64 \times 64$ in the first layer. You will not need this for both subproblems). It is constructed by 2 $5 \times 5$ convolution kernels. In general (e.g. a figure with height $h$ and width $w$ ), what is the receptive field of one pixel in layer 2? Please draw a 2D graph (excluding the channel dimension) to illustrate your answer.

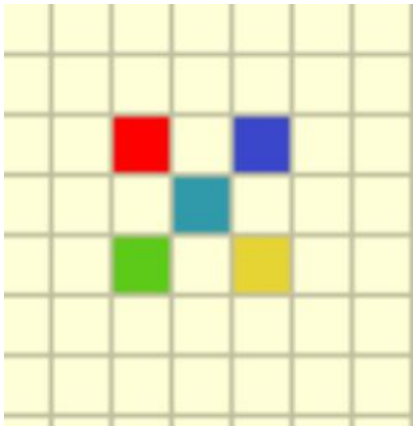
**Answer:**

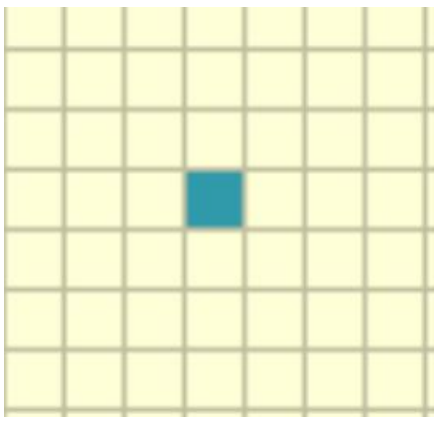
In layer 1: We have a 5x5 Convolution with no padding and stride of 1. This means that after this Convolution, each of the outputs will have a receptive field of 5.

In layer 2: We get the output of layer 1 in which each output had a receptive field of 5 and we apply a 5x5 Convolution with Padding = 2 and Stride = 2. For each pixel in the output of layer 2, each of the pixels involved in the 5x5 convolution has a receptive field

of 5. Since we had a stride of 1 in the previous layer, we will have overlaps in receptive fields of each of the pixels involved in the layer 2 convolution. Specifically each pixel receptive field overlaps in every column except 1. Thus the receptive field after layer 2 is 9. The receptive field quantity for each layer can be modeled then as the following equation: RfieldOut = RfieldPrev + (KernelSize - 1)*StridePrevLayer.
Thus our receptive field of a pixel in layer 2: **9**.


Input image


Output Layer 1 after 5x5 Conv w/ Stride = 1, Padding = 0

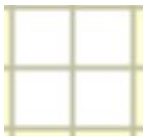

Output Layer 2 after 5x5 Conv w/ Stride = 2, Padding = 2

(b) Now consider that we add a n-by-n max-pooling layer following layer 2. In general (e.g. a figure with height $h$ and width $w$ ), what will be the receptive field after the max-pooling layer? Please also draw a 2D graph to illustrate your answer.

**Answer:**

From the logic in the previous part, we know that for some kernel size of n, the receptive field of a max pool operation after layer 2 is calculated as: 9 + (n-1)*2 = **(7 + 2n)**.

Output of layer2



Output max-pooling layer using kernelSize = 3, Stride = 2

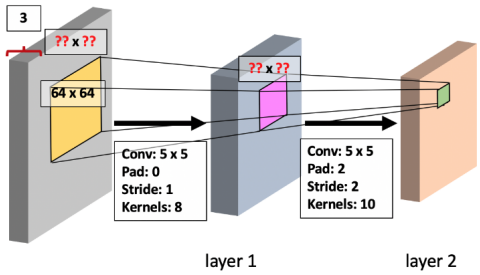



Figure 5: Convolutional Architecture.

For drawing the 2D graph, you may refer to Lecture 10 Slides 27.

3. Gradient Descent Optimization

   (a) Consider Figure 6, which depicts a cost function $L(x) : \mathbb{R}^2 \to \mathbb{R}$. The red dot represents the current estimate of $\mathbf{x}_t = [x_1, x_2]$ at time $t$. Please give a rough sketch of the direction of update steps that would be taken by vanilla SGD.

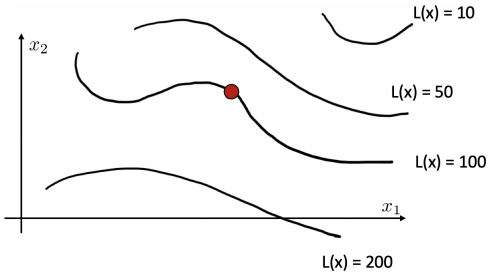
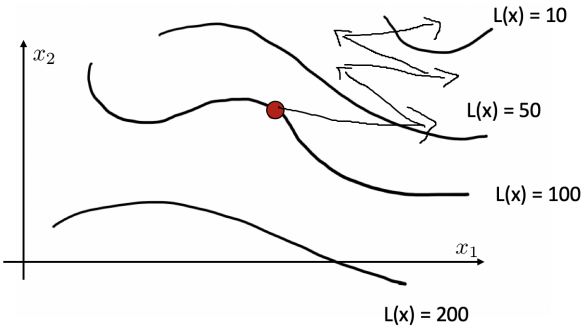



Figure 6: Contour lines of an arbitrary cost function with current estimate $\mathbf{x}_t$.

**Answer:**




As you can see vanilla SGD updates its gradient with every batch individually, thus each update has an equal weight in affecting the gradient. Therefore certain samples could move the gradient sub-optimally. We get this zig-zag pattern when trying to find the optimum.

(b) It is worth mentioning that the contour lines shown in Fig. 6 will change during optimization with SGD since the loss is evaluated over a single batch rather than the whole dataset. As discussed in class, this observation suggests that for unfortunate updates we might get stuck in saddle points, where the gradient of the vanilla SGD is 0. One way to tackle this problem is to use first and/or second order momentum. Please **briefly** explain why these moments are helpful and how they would change the update direction sketched in (a).

**Answer:**

Considering vanilla SGD, we know that taking gradient steps for each mini-batch individually could cause the updates to happen sub-optimally. Furthermore these updates could decide to take a random unbounded path that could take more time/iterations to reach the optimal solution. To alleviate this issue we can use first and second moments.
Using the first momentum: This acts basically like a velocity parameter that is multiplied to the gradient. Using this velocity(collected gradients from the previous iterations) to calculate the next gradient step allows for less vertical zig-zagging oscillation and instead opts for influencing the direction of the next gradient update so that it can make more progress horizontally towards the minima.
Using the second momentum: This does a similar direction tuning that the first momentum algorithm does except it does it in a slightly different way. This method instead alters the learning rate by dividing the learning rate by a squared gradient quantity like in RMSprop. This effectively achieves the same sort of zig-zag reduction, but tackles the problem of learning rate decay which can affect how much a certain gradient can affect an update.
The utilization of both first and second momentum allow for more versatility in optimization. ADAM utilizes both in order to calculate a gradient step. Utilizing both of them can allow for the benefits of having a momentum/velocity drive the direction of the gradient(especially useful when trapped in a sub-optimial minima), and a properly decaying learning rate. We can dampen the zig-zagging in the vertical direction as well as affect the amount of influence a certain gradient step has in an update step, by virtue of having an adapting learning rate as well.
In summary, having the moments in SGD will reduce the zig-zagging problem that SGD faces from having "bad" gradients and push more towards a direct gradient step towards the optimum. This typically will lead to faster convergence since the momentum will force gradients to be in the right direction instead of going in a direction away from the optimum.