Lab 3 Report

ECE-GY 9143 Intro to High Performance Machine Learning

Aditya Wagh (amw9425)

March 2022

Contents

1 Problem 1 **3**
- 1.1 Vanishing Gradients, Exploding Gradients, and Weight Initialization . . . 3
  - 1.1.1 Tanh and Sigmoid activation . . . 3
  - 1.1.2 ReLU with Xavier & He Initialization . . . 5
- 1.2 Investigation of Dying Network with ReLU activation . . . 6
- 1.3 Correction using Leaky ReLU activation . . . 6

2 Problem 2 **7**
- 2.1 Co-adaptation and Internal Co-variance Shift. . . . 7
- 2.2 Input Normalization with Hidden Layer Batch Normalization . . . 7
- 2.3 Batch Normalization on All Layers . . . 8
- 2.4 Dropout . . . 8
- 2.5 Dropout with Batch Normalization . . . 9

3 Problem 3 **10**
- 3.1 Finding maximum and minimum learning rate . . . 10
- 3.2 Train & Validation Loss & Accuracy Curves . . . 10
- 3.3 Experimentation with Increasing Batch Size . . . 11

4 Problem 4 **12**
- 4.1 Optimization Algorithms . . . 12
  - 4.1.1 Adagrad . . . 12
  - 4.1.2 Adadelta . . . 12
  - 4.1.3 Adam . . . 13
  - 4.1.4 Adam with Nesterov Correction . . . 13
  - 4.1.5 RMSProp . . . 14
  - 4.1.6 Comparison of AdaGrad, Adam and RMSProp . . . 14
- 4.2 Train Loss Comparison . . . 14
- 4.3 Dropout Train Loss Comparison . . . 15
- 4.4 Test Accuracy Comparison . . . 15

5 Problem 5 **16**
- 5.1 Calculations for the number of parameters in AlexNet . . . 16
- 5.2 Calculations for VGG19 . . . 18
- 5.3 Receptive Field . . . 18
  - 5.3.1 Proof for the receptive field of N convolution layers . . . 18
  - 5.3.2 Receptive field calculation . . . 19
- 5.4 Inception Net . . . 19
  - 5.4.1 Idea behind Inception Net . . . 19
  - 5.4.2 Output Size for Inception Net . . . 19
  - 5.4.3 Computational Complexity for Inception Net . . . 20
  - 5.4.4 Comparison of Naive and Bottleneck Version . . . 20

1 Problem 1

Google Colab Link

1.1 Vanishing Gradients, Exploding Gradients, and Weight Initialization

Vanishing Gradients is a phenomenon that is observed when training deep neural networks where gradients at a node computed during back-propagation turn out to be zero. This occurs when the gradients have very small values because of the nature of the activation functions and the magnitude of the inputs. Successive multiplication of a large number of small gradients (< 1) in the chain rule converges to zero. As a result, there are no weight updates and the neural network stops learning.

Similarly, successive multiplication of a large number of large gradients (> 1) in the chain rule causes the gradient value to be very large. This problem is termed the problem of exploding gradients, because of the large values of gradients thus obtained. This causes issues with convergence since the weight updates are very large.

Weight Initialization and usage of functions like Leaky ReLU help to counter this issue. If we consider the activation function tanh, it's derivative $(1 - \tanh^2(z))$ is dependent on the output of the neuron. Similarly for the sigmoid activation $\sigma(z)$ it's derivative $\sigma(z)(1 - \sigma(z))$ is also dependent on the output of the neuron. We don't want the outputs to be close to zero since gradients will be zero, making the network stop learning from data. The following figures try to explain how weight initialization helps with vanishing and exploding gradients.

1.1.1 Tanh and Sigmoid activation

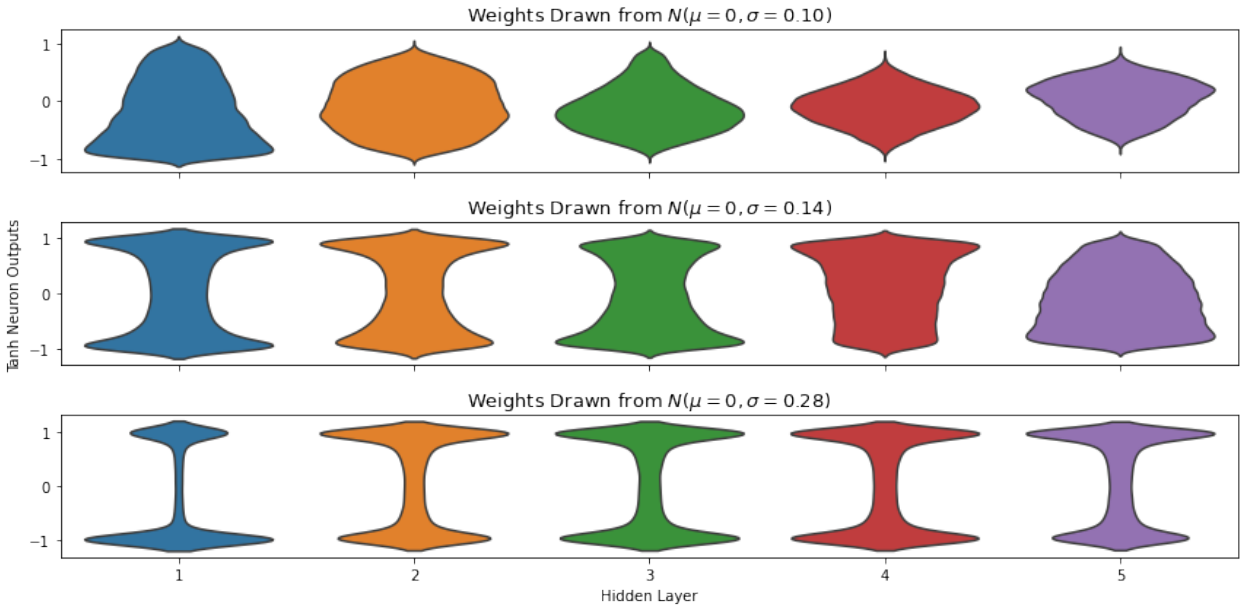

Figure 1: tanh and standard initialization

We can see that the neuron outputs are near -1 and 1. The derivative of tanh is $(1 - \tanh^2(z))$. If the neuron outputs are 1 and -1, this derivative will be zero. Hence, the chance for vanishing gradients increases immensely.

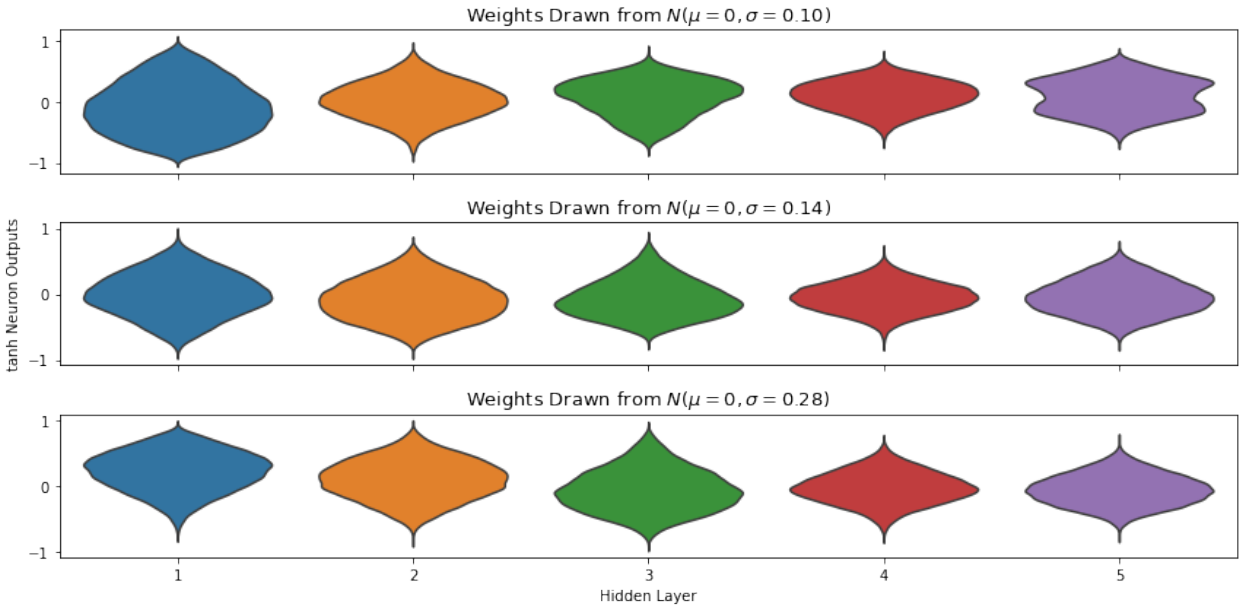

Figure 2: tanh and xavier initialization

When we use Xavier initialization, most of the neuron outputs are between -1 and 1, and thus the chance that the derivative of tanh is zero is reduces drastically. In fact, there are a large number of values near zero. Thus, we can see that Xavier initialization fixed the vanishing gradient issue in tanh.

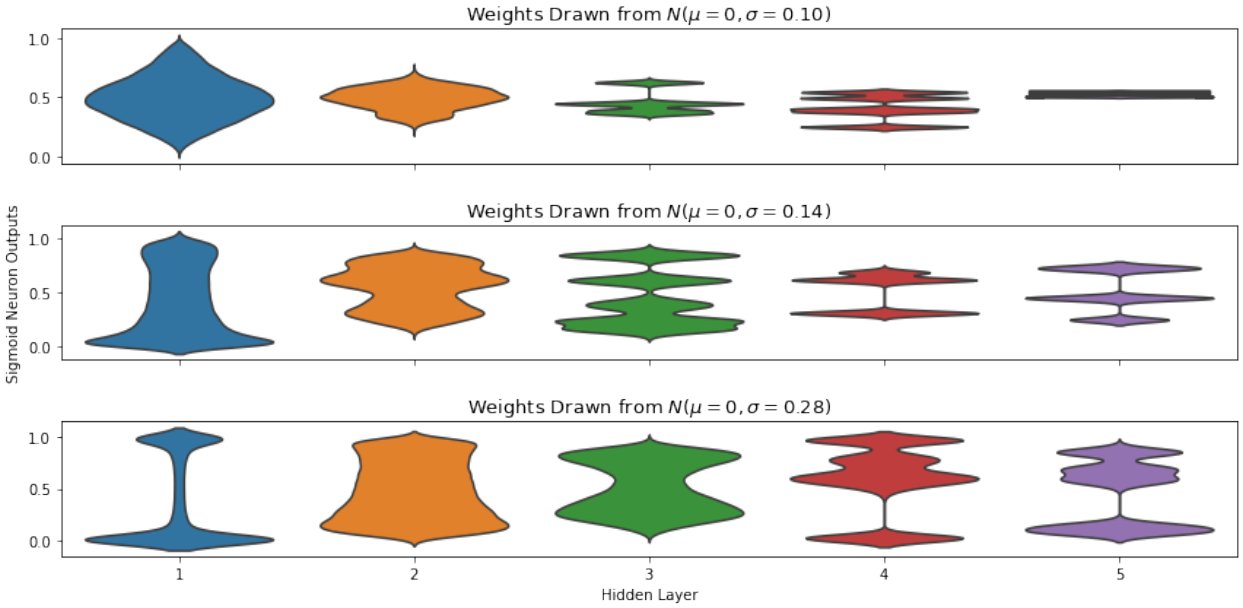

Figure 3: sigmoid and standard initialization

The derivative of sigmoid is $\sigma(z)(1 - \sigma(z))$. It will be zero when $\sigma(z) = 0 or \sigma(z) = 1$. Thus we want the output values to be far from 0 and 1. Hence, it is ideal if our values are close to 0.5. In Figure 3, we can see that a lot of values are either 0 or 1 when we use standard initialization.

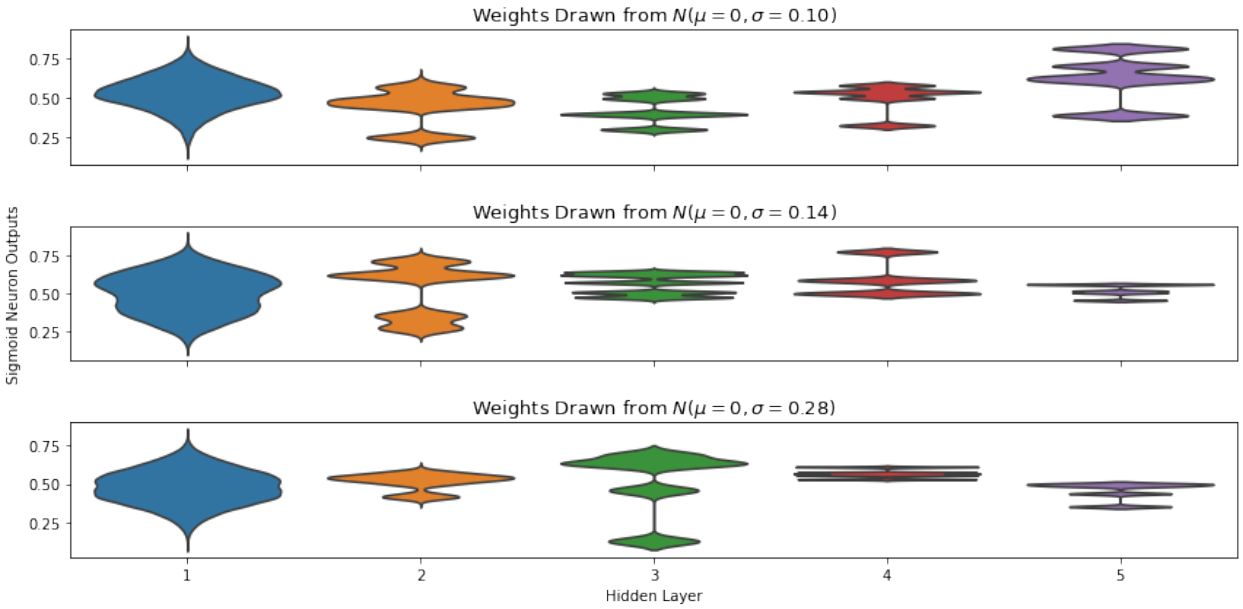

Figure 4: sigmoid and xavier initialization

Th Figure 4, we can see that most neuron outputs are between 0.25 and 0.75, which makes the gradients non zero, and also causes them to not explode.

1.1.2 ReLU with Xavier & He Initialization

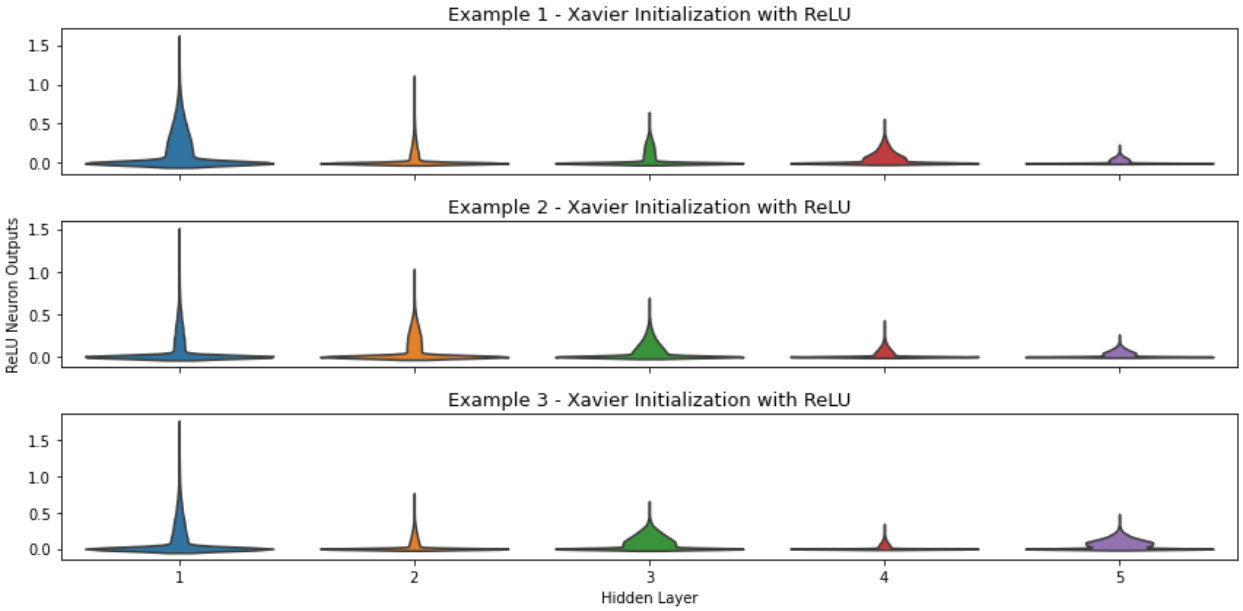

Figure 5: ReLU and xavier initialization

Here we can see that a lot of outputs are zero, which is not ideal for ReLU activation function. It causes the neurons to die.

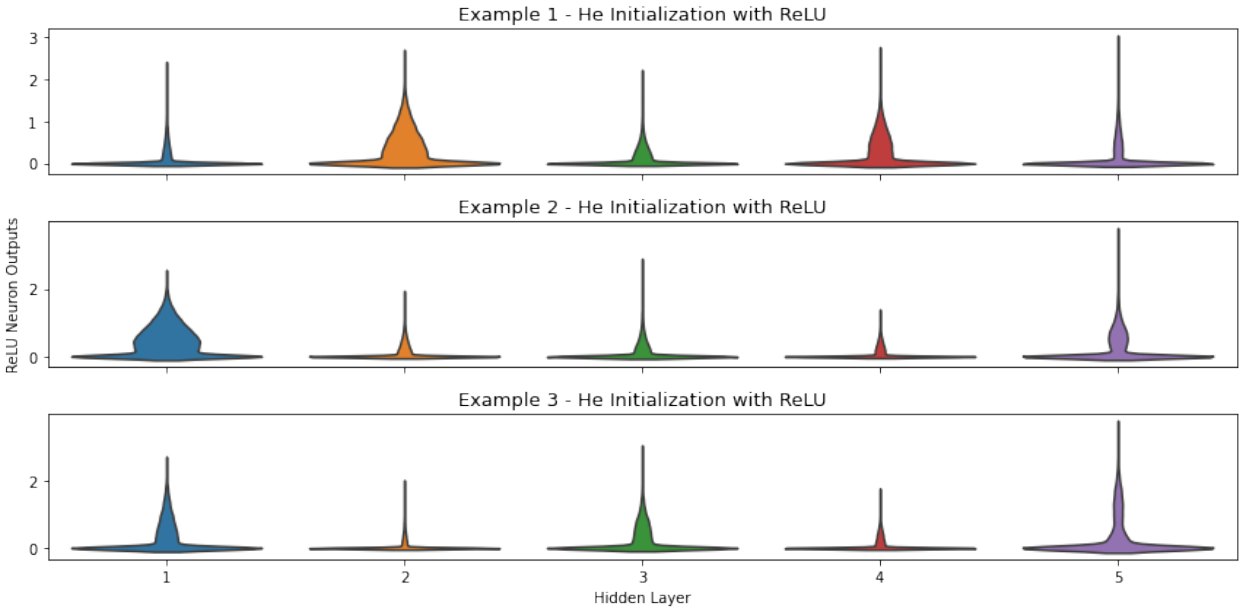

Figure 6: ReLU with He Initialization

Compared to Glorot initialization, the distribution in He initialization is much better. Hence the chance for dead neurons is decreased significantly.

1.2 Investigation of Dying Network with ReLU activation

The dying ReLU is a kind of vanishing gradient, which refers to a problem when ReLU neurons become inactive and only output 0 for any input. In the worst case of dying ReLU, ReLU neurons at a certain layer are all dead, i.e., the entire network dies and is referred to as the dying ReLU neural networks. To investigate this issue, we were supposed to generate 3000 random numbers between, and learn the function

$$f(x) = x \sin 5x$$

. We used a 10 hidden layer neural network with a width of 2 that is, 2 neurons in each layer. In out experiments, we did 1000 simulations where we generated data every time, between the range of $[-\sqrt{7}, \sqrt{7}]$. We used the ReLU activation function in out network. Each simulation was trained for 2 epochs. It turns out that with the ReLU function, **64.3% of the simulations resulted in a dying network for the function** $f(x) = x \sin 5x$**.** In the paper, they have experimented with the function $f(x) = |x|$ where, 90% of the simulations resulted in a dying network.

1.3 Correction using Leaky ReLU activation

It is observed that if we change the activation function to Leaky ReLU, there is **no scenario** in the 1000 simulations where the network is dead. This is the case for the function $f(x) = x \sin(5x)$

2 Problem 2

Google Colaboratory Notebook Link

2.1 Co-adaptation and Internal Co-variance Shift.

Deep neural networks are models composed of multiple layers of simple, non-linear neurons. With composition of enough neurons, the model can learn extremely complex functions that can accurately perform complicated tasks that are impossibly difficult to hard code, such as image classification, translation, speech recognition, etc. The key aspect of deep neural networks is that they are able to automatically learn data representation needed for features detection or classification without any a prior knowledge.

Often time, models with such complexity often can have exponentially many combinations of active neurons that can achieve high performance on the training set, but not all of them are robust enough to generalized well on unseen data (or testing data). This problem is also known as over-fitting.

One of the most prominent reasons for causing over-fitting is co-adaptation. According to wiki, at genetic level, co-adaptation is the accumulation of interacting genes in the gene pool of a population by selection. Selection pressures on one of the genes will affect its interacting proteins, after which compensatory changes occur. So in neural network, **co-adaptation** means that some neurons are highly dependent on others. If those independent neurons receive "bad" inputs, then the dependent neurons can be affected as well, and ultimately it can significantly alter the model performance, which is what might happen with over-fitting.

The simplest solution to over-fitting is dropout, and works as followed: each neuron in the network is randomly omitted with probability between 0 and 1.

Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities. **Internal Covariate Shift** is this change in the distribution of network activations due to the change in network parameters during training.

2.2 Input Normalization with Hidden Layer Batch Normalization

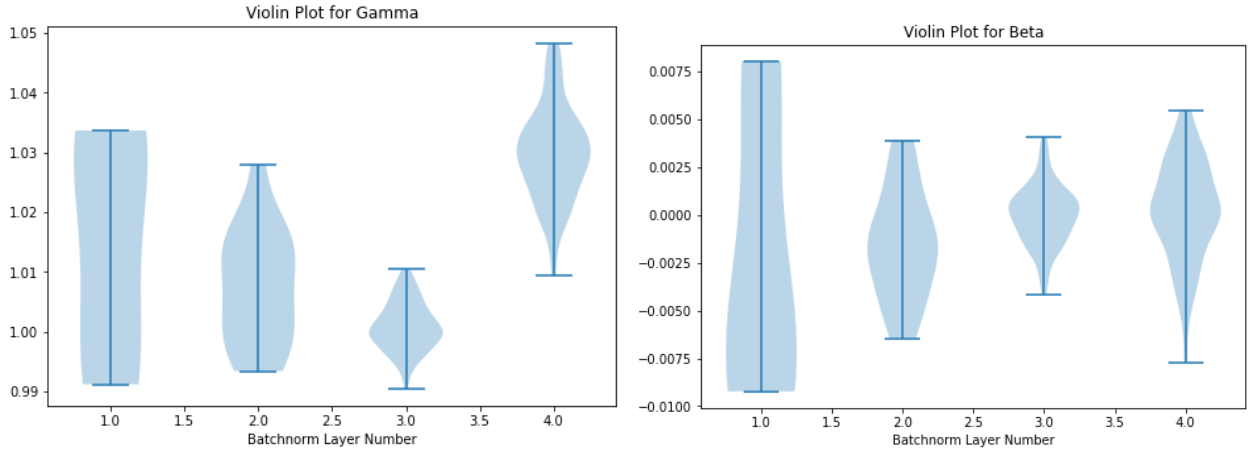

Figure 7: Batch Normalization parameters

2.3 Batch Normalization on All Layers

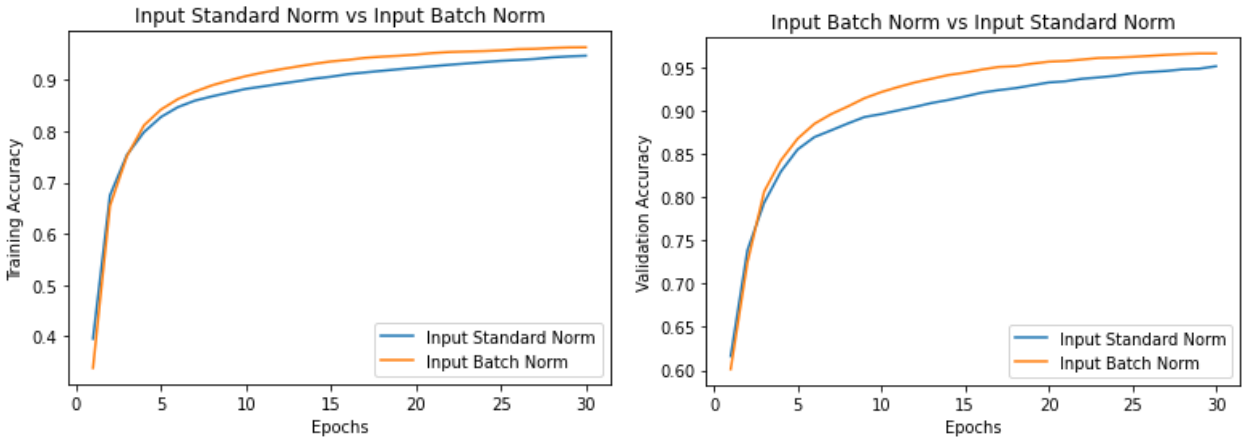

Figure 8: Training and Validation accuracy

As we can see, applying batch normalization to the input layer improved our training and validation accuracy compared to standard input normalization.

2.4 Dropout

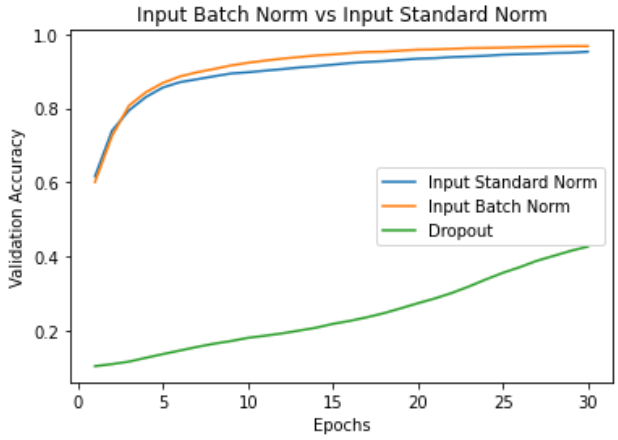

Figure 9: Validation Accuracy Comparison for Dropout, Input Standard Normalization and Input Batch Normalization

2.5 Dropout with Batch Normalization

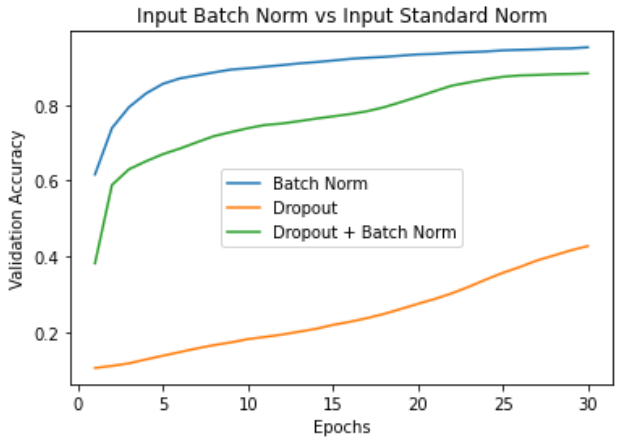

Figure 10: Validation Accuracy Comparison for Dropout, Dropout + Batch Norm and Batch Norm

It can be seen that using batch normalization with dropout improves the performance of the network compared to plain dropout.

3 Problem 3

Google Colaboratory Notebook Link

3.1 Finding maximum and minimum learning rate

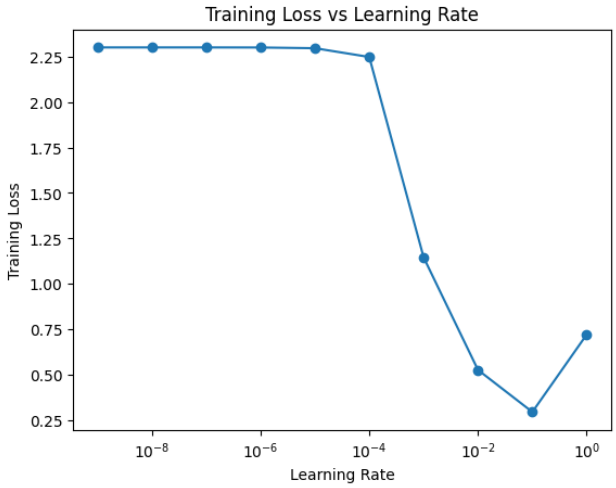

Figure 11: Candidate Learning rate experimentation

As per this graph, we can choose the max learning rate to be 0.1 and minimum learning rate to be 0.0001.

3.2 Train & Validation Loss & Accuracy Curves

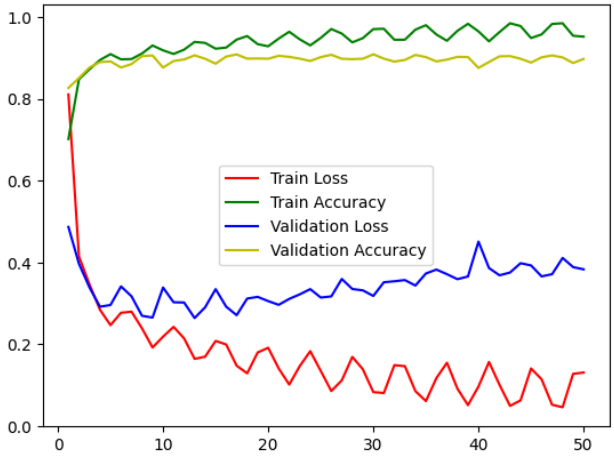

Figure 12: Loss and Accuracy Curves

3.3 Experimentation with Increasing Batch Size

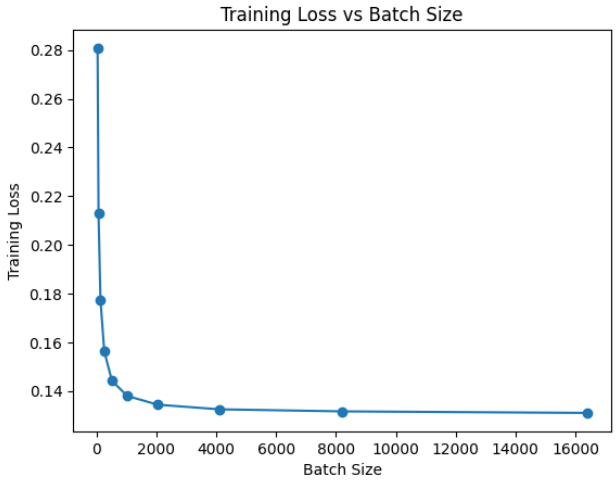

Figure 13: Experimentation with increasing batch size

If we compare the two training loss curves from part 1 with increasing batch size, we can say that the generalization is more or less the same.

4 Problem 4

Google Colaboratory Notebook Link

4.1 Optimization Algorithms

The following subsections showcase five different algorithms used for optimization in deep learning.

4.1.1 Adagrad

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay),
τ (initial accumulator value), η (lr decay)
initialize : $state\_sum_0 \leftarrow 0$

for $t = 1$ **to** ... **do**
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
$\tilde{\gamma} \leftarrow \gamma/(1 + (t-1)\eta)$
if $\lambda \neq 0$
$g_t \leftarrow g_t + \lambda\theta_{t-1}$
$state\_sum_t \leftarrow state\_sum_{t-1} + g_t^2$
$\theta_t \leftarrow \theta_{t-1} - \tilde{\gamma}\dfrac{g_t}{\sqrt{state\_sum_t} + \epsilon}$

return θ_t

Figure 14: Adagrad algorithm

4.1.2 Adadelta

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), ρ (decay), λ (weight decay)
initialize : $v_0 \leftarrow 0$ (square avg), $u_0 \leftarrow 0$ (accumulate variables)

for $t = 1$ **to** ... **do**
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
$if\ \lambda \neq 0$
$g_t \leftarrow g_t + \lambda\theta_{t-1}$
$v_t \leftarrow v_{t-1}\rho + g_t^2(1-\rho)$
$\Delta x_t \leftarrow \dfrac{\sqrt{u_{t-1} + \epsilon}}{\sqrt{v_t + \epsilon}} g_t$
$u_t \leftarrow u_{t-1}\rho + \Delta x_t^2(1-\rho)$
$\theta_t \leftarrow \theta_{t-1} - \gamma\Delta x_t$

return θ_t

Figure 15: Adadelta Algorithm

4.1.3 Adam

input : γ (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective)
λ (weight decay), *amsgrad*, *maximize*
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment), $\widehat{v_0}^{max} \leftarrow 0$

for $t = 1$ **to** ... **do**
  if *maximize* :
    $g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1})$
  else
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  if $\lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow m_t / \big(1-\beta_1^t\big)$
  $\widehat{v_t} \leftarrow v_t / \big(1-\beta_2^t\big)$
  if *amsgrad*
    $\widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t})$
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}^{max}} + \epsilon\big)$
  else
    $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}} + \epsilon\big)$

return θ_t

Figure 16: Adam Algorithm

4.1.4 Adam with Nesterov Correction

input : γ_t (lr), β_1, β_2 (betas), θ_0 (params), $f(\theta)$ (objective)
λ (weight decay), ψ (momentum decay)
initialize : $m_0 \leftarrow 0$ (first moment), $v_0 \leftarrow 0$ (second moment)

for $t = 1$ **to** ... **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
  $if\ \lambda \neq 0$
    $g_t \leftarrow g_t + \lambda\theta_{t-1}$
  $\mu_t \leftarrow \beta_1\big(1 - \frac{1}{2} 0.96^{t\psi}\big)$
  $\mu_{t+1} \leftarrow \beta_1\big(1 - \frac{1}{2} 0.96^{(t+1)\psi}\big)$
  $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
  $\widehat{m_t} \leftarrow \mu_{t+1} m_t / (1 - \prod_{i=1}^{t+1} \mu_i)$
    $+ (1-\mu_t) g_t / (1 - \prod_{i=1}^{t} \mu_i)$
  $\widehat{v_t} \leftarrow v_t / \big(1-\beta_2^t\big)$
  $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}} + \epsilon\big)$

return θ_t

Figure 17: Adam Algorithm with Nesterov Correction

4.1.5 RMSProp

input : α (alpha), γ (lr), θ_0 (params), $f(\theta)$ (objective)
λ (weight decay), μ (momentum), *centered*
initialize : $v_0 \leftarrow 0$ (square average), $\mathbf{b}_0 \leftarrow 0$ (buffer), $g_0^{ave} \leftarrow 0$

for $t = 1$ **to** ... **do**
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
if $\lambda \neq 0$
$g_t \leftarrow g_t + \lambda\theta_{t-1}$
$v_t \leftarrow \alpha v_{t-1} + (1-\alpha)g_t^2$
$\tilde{v}_t \leftarrow v_t$
if centered
$g_t^{ave} \leftarrow g_{t-1}^{ave}\alpha + (1-\alpha)g_t$
$\tilde{v}_t \leftarrow \tilde{v}_t - \left(g_t^{ave}\right)^2$
if $\mu > 0$
$\mathbf{b}_t \leftarrow \mu\mathbf{b}_{t-1} + g_t/\left(\sqrt{\tilde{v}_t} + \epsilon\right)$
$\theta_t \leftarrow \theta_{t-1} - \gamma\mathbf{b}_t$
else
$\theta_t \leftarrow \theta_{t-1} - \gamma g_t/\left(\sqrt{\tilde{v}_t} + \epsilon\right)$

return θ_t

Figure 18: RMSProp Algorithm

4.1.6 Comparison of AdaGrad, Adam and RMSProp

RMSprop is an extension of Adagrad that deals with its radically diminishing learning rates. It is identical to Adadelta, except that Adadelta uses the RMS of parameter updates in the numinator update rule. Adam, finally, adds bias-correction and momentum to RMSprop. RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances. Bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser.

4.2 Train Loss Comparison

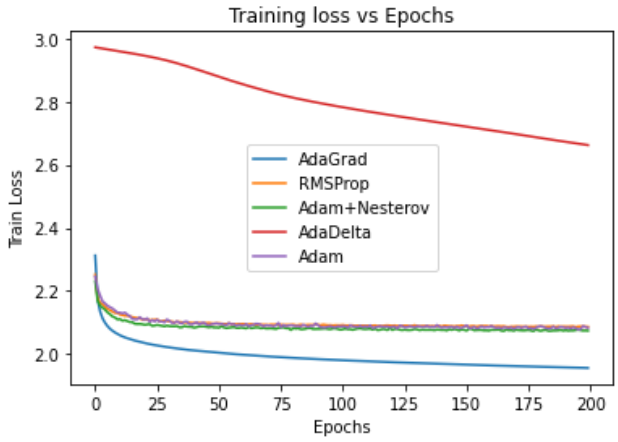

Figure 19: L2 Training Loss

It can be seen that AdaGrad has the least training loss among the methods.

4.3 Dropout Train Loss Comparison

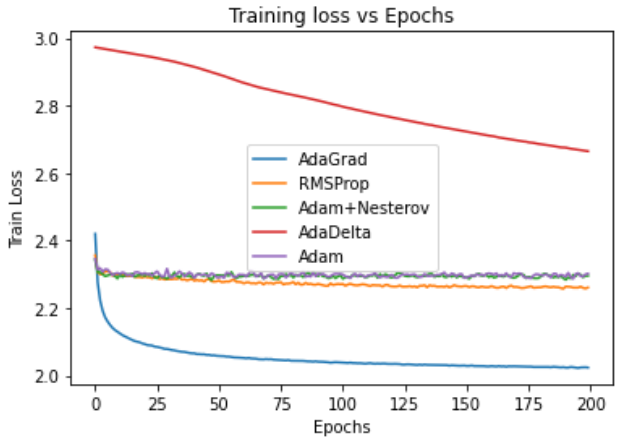

Figure 20: L2 + Dropout Training Loss

Optimizer	Training Time L2(s)	Training Time L2 + Dropout(s)
Adagrad	938.8822618611157	928.891880008392
RMSProp	921.0949673959985	920.1396476849914
Adam	919.4897332647815	918.4741656240076
NAdam	919.6869753077626	918.9006990632042
Adadelta	919.4185678204522	1837.0741803962737

Table 1: Training Time for 200 Epochs

We can see that dropout with L2 regularization takes slightly less time to train compared to L2 regularization only.

4.4 Test Accuracy Comparison

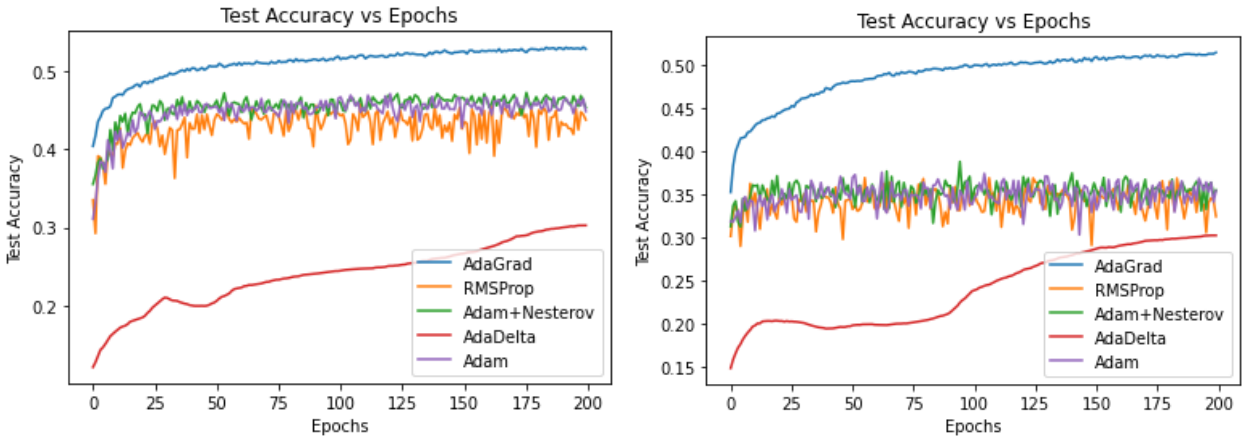

Figure 21: L2 Regularization (Left), L2 + Dropout Regularization (Right)

5 Problem 5

5.1 Calculations for the number of parameters in AlexNet

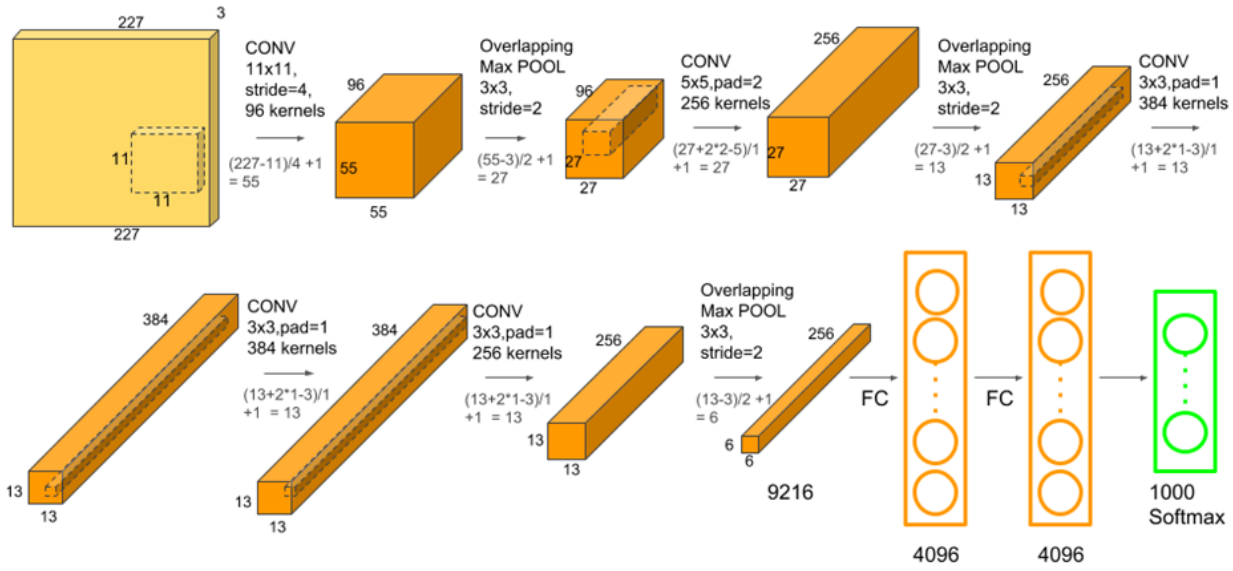

Figure 22: AlexNet Architecture

The architecture of AlexNet is shown in the figure above. As we can see, it has 11 layers in total. There are 5 convolution layers, 3 Max Pooling layers and 3 Fully Connected Layers. The size of the input images is (3 * 227 * 227) where 3 represents the number of channels in the image, and 224 is the width and height of the image.

For a convolutional layer, the number of learnable parameters is (F * F * C) * K + B, where F is the size of the filter, C is the number of input channels, K is the number of filters and B which is the bias for each filter, which is equal to K. For a linear layer, the number of learnable parameters is I * O + B, where I is the number of input neurons, and O is the number of output neurons, and B is the number of biases in the layer, which is equal to the number of output neurons.

Therefore, for the first convolutional layer, the number of learnable parameters are (11 * 11 * 3) * 96 + 96 = 34,848 + 96 = 34944. Here 96 represents the number of convolutional filters, and (3 * 11 * 11) represents the size of the filters. Since there are 3 channels in the input image, each filter would be of the size (3 * 11 * 11). The output size after a convolution operation is

$$\lfloor \frac{N + 2P - F}{S} + 1 \rfloor$$

For the first pooling layer, there are no learnable parameters. For the second convolutional layer, the number of learnable parameters are (5 * 5 * 96) * 256 + 256= 614,656. For the third convolution layer, the number of learnable parameters are (3 * 3 * 256) * 384 + 384= 885,120. For the fourth convolution layer, the number of learnable parameters are (3 * 3 * 384) * 384 + 384 = 1,327,488. For the fifth convolution layer, the number of learnable parameters are (3 * 3 * 384) * 256 + 256 = 884,992. The output size after

all the convolution operations can be calculated using the formula mentioned above. The size of the output comes out to be (5 * 5 * 256). This output is then flattened out to feed it into the fully connected layer. Hence, the number of parameters in the first fully connected layer are to be (5 * 5 * 256) * 4096 + 4096 = 37,752,832. The second fully connected layer has 4096 units, therefore the learnable parameters in the second fully connected layer are (4096 * 4096) + 4096 = 16,781,312. The next, and final fully connected layer has 1000 units, therefore the number of learnable parameters are (4096 * 1000) + 1000 = 4,097,000. The sum of all parameters is 34,944 + 0 + 614,656 + 0 + 885,120 + 1,327,488 + 884,992 + 0 + 37,752,832 + 16,781,312 + 4,097,000 = 62,378,344

All the results are summarized in the table below. The number of parameters contains the number of weights + the number of biases, as we have calculated above.

AlexNet Network - Structural Details													
Input			Output			Layer	Stride	Pad	Kernel size		in	out	# of Param
227	227	3	55	55	96	conv1	4	0	11	11	3	96	34944
55	55	96	27	27	96	maxpool1	2	0	3	3	96	96	0
27	27	96	27	27	256	conv2	1	2	5	5	96	256	614656
27	27	256	13	13	256	maxpool2	2	0	3	3	256	256	0
13	13	256	13	13	384	conv3	1	1	3	3	256	384	885120
13	13	384	13	13	384	conv4	1	1	3	3	384	384	1327488
13	13	384	13	13	256	conv5	1	1	3	3	384	256	884992
13	13	256	6	6	256	maxpool5	2	0	3	3	256	256	0
						fc6			1	1	9216	4096	37752832
						fc7			1	1	4096	4096	16781312
						fc8			1	1	4096	1000	4097000
Total													**62,378,344**

Figure 23: AlexNet Parameter Calculations

5.2 Calculations for VGG19

Layer	Number of activations (memory)	Parameters (compute)
Input	3 * 224 * 224 = 150,528	0
Conv3-64	64 * 224 * 224 = 3,211,264	(3 * 3 * 3) * 64 = 1728
Conv3-64	64 * 224 * 224 = 3,211,264	(64 * 3 * 3) * 64 = 36,864
Pool2	64 *112 * 112 = 802,816	0
Conv3-128	128 * 112 * 112 = 1,605,632	(64 * 3 * 3) * 128 = 73,728
Conv3-128	128 * 112 * 112 = 1,605,632	(128 * 3 * 3) * 128 = 147,456
Pool2	128 * 56 * 56 = 401,408	0
Conv3-256	256 * 56 * 56 = 802,816	(128 * 3 * 3) * 256 = 294,912
Conv3-256	256 * 56 * 56 = 802,816	(256 * 3 * 3) * 256 = 589,824
Conv3-256	256 * 56 * 56 = 802,816	(256 * 3 * 3) * 256 = 589,824
Conv3-256	256 * 56 * 56 = 802,816	(256 * 3 * 3) * 256 = 589,824
Pool2	256 * 28 * 28 = 200,704	0
Conv3-512	512 * 28 * 28 = 401,408	(256 * 3 * 3) * 512 = 1,179,648
Conv3-512	512 * 28 * 28 = 401,408	(512 * 3 * 3) * 512 = 2,359,296
Conv3-512	512 * 28 * 28 = 401,408	(512 * 3 * 3) * 512 = 2,359,296
Conv3-512	512 * 28 * 28 = 401,408	(512 * 3 * 3) * 512 = 2,359,296
Pool2	512 * 14 * 14 = 100,352	0
Conv3-512	512 * 14 * 14 = 100,352	(512 * 3 * 3) * 512 = 2,359,296
Conv3-512	512 * 14 * 14 = 100,352	(512 * 3 * 3) * 512 = 2,359,296
Conv3-512	512 * 14 * 14 = 100,352	(512 * 3 * 3) * 512 = 2,359,296
Conv3-512	512 * 14 * 14= 100,352	(512 * 3 * 3) * 512 = 2,359,296
Pool2	512 * 7 * 7 = 25,088	0
FC	4096	(512 * 7 * 7) * 4096 = 102,760,448
FC	4096	4096 * 4096 = 16,77,216
FC	1000	1000 * 4096 = 4,096,000
Total		143,667,240

5.3 Receptive Field

5.3.1 Proof for the receptive field of N convolution layers

The receptive field r of a N layer convolutional network with filter size F and stride S at each layer is given by

$$r = \sum_{l=1}^{N} \left((F-1) \prod_{i=1}^{N-1} S \right) + 1$$

In VGG network, the stride for a convolution layer is 1. With this insight, the equation above reduces to

$$r = \sum_{l=1}^{N} ((F-1)) + 1$$

Upon further simplification, we can see that, $r = N\,(F-1) + 1 = NF - N + 1$

For a convolution layer with filter size NF – N + 1 and stride 1, the receptive field is

$$r = \sum_{l=1}^{1} (((NF - N + 1) - 1)) + 1$$

$$r = (NF - N) + 1 = \ NF \ - \ N \ + \ 1$$

Thus we can see that, for a stack of N convolutional layers with filter size F and stride of 1, the receptive field is equal to a single convolutional layer with a filter size of (NF – N + 1) and stride 1.

5.3.2 Receptive field calculation

Using the results mentioned above, the receptive field of 3 filters of size 5x5 is $r = (NF - N + 1) = 3 * 5 - 3 + 1 = 15 - 3 + 1 = 13$

5.4 Inception Net

5.4.1 Idea behind Inception Net

Inception Modules are used in Convolutional Neural Networks to allow for more efficient computation and deeper Networks through a dimensionality reduction with stacked 1×1 convolutions. The modules were designed to solve the problem of computational expense, as well as overfitting, among other issues. The solution, in short, is to take multiple kernel filter sizes within the CNN, and rather than stacking them sequentially, ordering them to operate on the same level.

5.4.2 Output Size for Inception Net

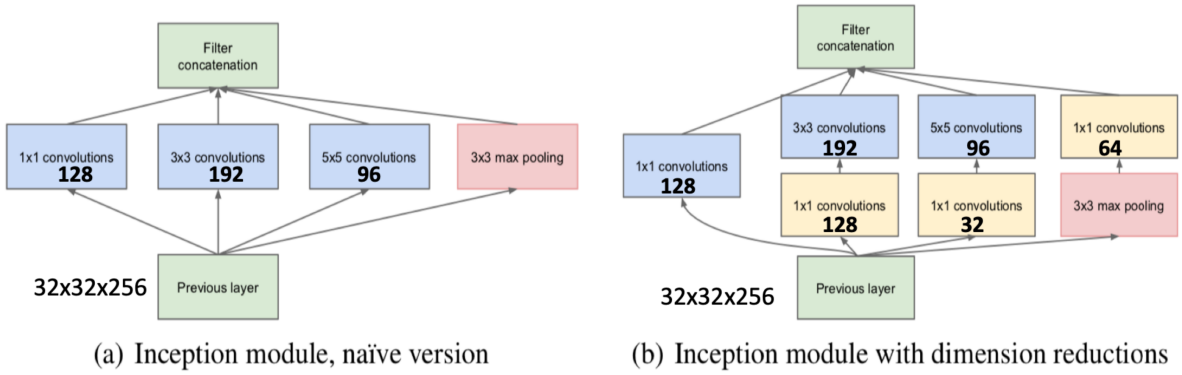

Figure 24: Inception Modules

For the naive architecture, the size of output after filter concatenation will be 32 x 32 x (128 + 192 + 96 + 256) = 32 x 32 x 672. The output feature width and height is same as the input width and height in inception net. The filters are concatenated that's why we add the number of channels from each convolution layer.

For the dimensionality reduction inception architecture, in the 1x1 convolution layer, the output dimensions would be 32 x 32 x 128 since there are 128 filters. For the 3x3 convolutions with 1x1 convolutions, the output of 1 x 1 convolutions is 32 x 32 x 128, and the output of 3 x 3 convolutions is 32 x 32 x 192. Similarly, for 5x5 convolutions with 1x1 convolutions, the output of bottleneck is 32 x 32 x 32, and the output of 5x5 convolutions is 32 x 32 x 96. For the 3x3 max pool, the output is 32 x 32 x 256. The output of the 1x1 convolution after pooling is 32 x 32 x 64, since there are 64 such filters. After concatenating the output of all layers, the final output dimension is 32 x 32 x (128 + 192 + 96 + 64) = 32 x 32 x 480.

5.4.3 Computational Complexity for Inception Net

For the naive version, the number of computations in the 1x1 conv layer are (32 x 32 x 128) x (1 x 1 x 256) = 33,554,432 FLOP. Similarly, for the 3x3 convolution, number of computations are (32 x 32 x 192) x (3 x 3 x 256) = 452,984,832 and for the 5x5 convolution, number of computations are (32 x 32 x 96) x (5 x 5 x 256) = 629,145,600. The total number of computations are 1,115,684,864 $\approx$ 1.1 Billion FLOP.

For the dimensionality reduction inception architecture, the number of computations in the 1x1 conv layer are (32 x 32 x 128) x (1 x 1 x 256) = 33,554,432. For the 3x3 and 1x1 bottleneck, we will compute operations step by step. The number of computations are (32 x 32 x 128) x (1 x 1 x 256) + (32 x 32 x 192) x (3 x 3 x 128) = 260,046,848. Similarly, for the 5x5 and 1x1 convolution, we get the number of computations as (32 x 32 x 32) x (1 x 1 x 256) + (32 x 32 x 96) x (5 x 5 x 32) = 87,031,808. For the last 1x1 conv layer with pooling, the number of convolutions are (32 x 32 x 64) * (1 x 1 x 256) = 16,777,216. The total number of computations are 397,410,304 $\approx$ 400M FLOP.

5.4.4 Comparison of Naive and Bottleneck Version

As we can see, the naive version has 1.1 Billion computations, where as the dimensionality reduction architecture has 400M computations. Even for modern computers, 1B computations are a lot! Thus, the dimentionality reduction architecture helps to reduce the computational complexity of the model. Compared to the naive architecture, The number of computations are cut to 35% of the original amount. That's 65% savings in the number of computations!