**COMPUTER VISION ASSIGNMENT\_3**

**1.After each stride-2 conv, why do we double the number of filters?**

After each stride-2 convolution in a convolutional neural network (CNN), it is common to double the number of filters in the following layer for a number of reasons:

Increased representation: Doubling the number of filters increases the number of feature maps produced by the convolutional layer, which in turn provides a more complete representation of the input image. This increased representation can help the network to capture more information about the image, and make more accurate predictions.

Improved abstraction: As the network increases the number of filters, it also increases the level of abstraction, as each filter can learn to detect different, increasingly complex patterns in the input image. This can help the network to learn more invariant features that are not sensitive to small changes in the input, such as translations or rotations.

Control over computation: Increasing the number of filters can help control the computational cost of the network, as the increased number of filters can help reduce overfitting, which occurs when a network memorizes the training data instead of learning to generalize to new data.

However, it is important to note that increasing the number of filters also increases the computational cost of the network, as the network needs to perform more computations for each layer. As a result, finding the right balance between the number of filters and the computational cost is an important part of designing and training effective CNNs.

**2. Why do we use a larger kernel with MNIST (with simple cnn) in the first conv?**

When using a convolutional neural network (CNN) to classify images of the MNIST dataset, it is common to use a larger kernel in the first convolutional layer for several reasons:

Image resolution: The MNIST images are 28x28 pixels, which are relatively small compared to many other image datasets. Using a larger kernel in the first layer can help to capture larger features in the image, such as strokes and shapes, that may be important for classifying the image.

Feature extraction: The first layer of a CNN is often responsible for extracting lower-level features from the image, such as edges, corners, and textures. Using a larger kernel in the first layer can help to extract these features more effectively, and make them available for the later layers to use for more complex feature extraction and classification.

Computational efficiency: Using a larger kernel in the first layer can reduce the number of parameters in the network, which can help to control the computational cost of the network. This can be particularly important for smaller datasets like MNIST, where overfitting can be a problem, as it can help to reduce the risk of overfitting by making the network less complex.

It is important to note that the choice of kernel size is not set in stone and can vary based on the specific requirements of the task and the size of the input images. In some cases, using a smaller kernel in the first layer may be more effective, as it can help to capture fine-grained details in the image that may be important for classification.

**3. What data is saved by ActivationStats for each layer?**

ActivationStats is a tool used in deep learning to monitor the activations of neurons in a neural network during training. For each layer, ActivationStats saves the following data:

Mean activation: The mean activation value across all neurons in the layer.

Standard deviation: The standard deviation of the activation values across all neurons in the layer.

Minimum activation: The minimum activation value across all neurons in the layer.

Maximum activation: The maximum activation value across all neurons in the layer.

Histogram of activations: A histogram of the activation values across all neurons in the layer, showing the distribution of activations.

This data can be used to understand the behavior of the activations in the network, such as identifying neurons that are dead (i.e., always producing the same output) or neurons that are saturating (i.e., producing activations close to the maximum or minimum value). By monitoring the activations, it is possible to identify problems in the network architecture or training process and make adjustments to improve performance.

**4. How do we get a learner’s callback after they’ve completed training?**

In deep learning, a callback is a function that is executed at specific points during training, such as after each epoch, after each batch, or after training has completed. To get a learner's callback after they have completed training, you can use the Learner.callback\_fns attribute in the deep learning library you are using, such as fastai or PyTorch.

Here is an example using fastai:

from fastai.basic\_train import Learner

from fastai.callback import Callback

class MyCallback(Callback):

def on\_epoch\_end(self, \*\*kwargs):

print("Training has completed!")

learn = Learner(..., ..., ...)

learn.callback\_fns.append(MyCallback)

learn.fit(epochs=10)

In this example, the MyCallback class is defined as a subclass of Callback, and its on\_epoch\_end method is overridden to print "Training has completed!" after each epoch. The Learner is then created, and the MyCallback is added to the Learner.callback\_fns list. When the learn.fit method is called, the MyCallback will be executed at the end of each epoch, and the message "Training has completed!" will be printed.

A similar process can be used in PyTorch by creating a custom callback class and attaching it to the training loop using the torch.nn.Module.register\_forward\_hook method.

**5. What are the drawbacks of activations above zero?**

Activations above zero in a neural network can have several drawbacks, including:

Saturation: If the activations become too large, they can saturate the activation function and become stuck at the maximum or minimum value, which can cause the gradient to vanish during backpropagation and prevent the model from learning.

Overfitting: Large activations can indicate overfitting, where the model is memorizing the training data instead of learning general features that can be used to make predictions on new data.

Exploding gradients: If the activations become too large, they can cause the gradients to explode during backpropagation, which can result in numerical instability and cause the model to fail to converge.

Computational efficiency: Large activations can result in increased computational cost and memory usage, as well as reduced numerical stability, which can make training more difficult and slow.

To mitigate these drawbacks, it is often necessary to use techniques such as weight decay, dropout, or activation clipping to regularize the model and prevent the activations from becoming too large. Additionally, it may be necessary to adjust the architecture of the model or the training process to improve its generalization ability and prevent overfitting.

**6.Draw up the benefits and drawbacks of practicing in larger batches?**

Benefits of using larger batches in deep learning:

Increased computational efficiency: Using larger batches can reduce the amount of time required to process each batch and make training faster, especially when using GPUs, which are optimized for processing large arrays of data.

Improved generalization: Using larger batches can reduce the variance of the gradient estimates and result in a more stable optimization process, leading to improved generalization performance.

Better parallelization: Larger batches can be divided into smaller chunks that can be processed in parallel, making it possible to train models on larger datasets and on multiple GPUs.

Drawbacks of using larger batches:

Increased memory usage: Larger batches require more memory to store the activations, weights, and gradients, which can be a limiting factor when training large models.

Reduced statistical efficiency: Using larger batches can result in reduced statistical efficiency, as the gradient estimates are based on fewer samples per update. This can result in slower convergence and longer training times.

Batch normalization issues: Batch normalization can become less effective with larger batches, as the mean and variance of the activations may be less representative of the entire dataset. This can result in reduced performance and increased instability during training.

In general, the optimal batch size will depend on the size and complexity of the model, the computational resources available, and the desired trade-off between training speed and generalization performance. Practical recommendations for batch size are often in the range of 32-512. It is common to experiment with different batch sizes to find the best balance between speed and performance.

**7. Why should we avoid starting training with a high learning rate?**

Starting training with a high learning rate can be problematic for several reasons:

Poor convergence: If the learning rate is too high, the model parameters may oscillate or diverge instead of converging to a good solution. This can result in poor performance and failure to learn the desired task.

Slow convergence: Even if the learning rate is not too high to cause divergence, it can still slow down convergence if it is too large. This can result in longer training times and increased computational cost.

Overstepping optimal solution: With a high learning rate, the model parameters may overshoot the optimal solution and settle in a suboptimal region, causing the model to underperform.

Stochastic gradient noise: When using mini-batch gradient descent, the gradient estimates can be noisy and fluctuate from one iteration to the next. With a high learning rate, the model parameters may be updated too aggressively and be sensitive to this noise, resulting in poor performance and instability.

To avoid these problems, it is common to start training with a low learning rate and gradually increase it over time. This allows the model to converge to a good solution gradually and reduces the risk of overstepping the optimal solution or being affected by noisy gradients. Additionally, learning rate schedules, such as step decay, cyclical learning rate, or dynamic learning rate, can be used to adjust the learning rate during training based on the performance of the model and the convergence progress.

**8. What are the pros of studying with a high rate of learning?**

There are not many benefits to starting with a high learning rate in deep learning. However, here are a few potential advantages:

Faster convergence: In some cases, a high learning rate can lead to faster convergence, especially when the optimization problem is well-conditioned and the gradients are relatively stable. This can result in shorter training times and reduced computational cost.

Escaping local minima: When training deep neural networks, it is common to encounter local minima, where the model parameters get stuck in a suboptimal solution. With a high learning rate, the model may have a higher chance of escaping these local minima and finding a better solution.

Simpler learning rate schedules: Using a high learning rate can simplify the learning rate schedule, as the model may converge faster without the need for multiple learning rate changes or complex schedules.

Overall, the drawbacks of starting with a high learning rate, such as poor convergence, slow convergence, overstepping the optimal solution, and sensitivity to stochastic gradient noise, outweigh the benefits. It is generally recommended to start with a low learning rate and gradually increase it over time, or to use a learning rate schedule, such as step decay or cyclical learning rate, to adjust the learning rate during training.

**9. Why do we want to end the training with a low learning rate?**

Ending training with a low learning rate is useful for several reasons:

Fine-tuning: As the model approaches convergence, the gradients become smaller and the updates to the model parameters become smaller as well. This can help the model to fine-tune the parameters and converge to a good solution with high accuracy.

Stabilizing the convergence: A low learning rate can help to stabilize the convergence, especially when the model is close to the optimal solution. This can prevent overshooting the optimal solution or being affected by noisy gradients.

Preventing overfitting: With a low learning rate, the model parameters are updated more slowly and the model is less likely to overfit the training data, especially if the model is already overfitting.

Improving generalization: A low learning rate can help the model to generalize better to new data by preventing overfitting and allowing the model to fine-tune the parameters to a good solution.

In summary, ending training with a low learning rate can help to stabilize the convergence, prevent overfitting, and improve the generalization performance of the model. However, it is also important to choose a learning rate that is not too low, as this can slow down the convergence or prevent the model from finding a good solution.