**Assignment-3**

**Q.1) Explain batch Normalization, Instance Normalization and group Normalization with its Proper use?**

* Batch Normalization, Instance Normalization, and Group Normalization are normalization techniques used in deep learning models to improve training stability, speed up convergence, and enhance model performance. Here's an explanation of each technique and their proper use:

1. **Batch Normalization (BN):** Batch Normalization is a technique that normalizes the activations of each layer in a neural network by normalizing the mini-batch statistics. It aims to address the internal covariate shift problem, where the distribution of inputs to each layer of the network changes during training. BN calculates the mean and variance of the mini-batch and normalizes the activations using these statistics. It introduces learnable parameters (scale and shift) to allow the model to adapt the normalized activations.

* **Proper Use:** Batch Normalization is commonly used in deep neural networks, especially in convolutional neural networks (CNNs). It can be applied after the convolutional and fully connected layers in the network. BN is effective in accelerating training convergence, reducing the dependence on careful initialization, and providing regularization effects. It helps stabilize training, improve generalization, and allows the use of higher learning rates.

1. **Instance Normalization (IN):** Instance Normalization is a technique that normalizes the activations within each individual sample in a mini-batch. It calculates the mean and variance for each sample independently, regardless of other samples in the mini-batch. IN aims to address style variations across different samples by normalizing the activations within each instance.

* **Proper Use:** Instance Normalization is commonly used in style transfer applications, where the goal is to transfer the style of one image onto another. It helps to remove instance-specific statistics and encourages the network to focus on capturing the style information. IN is also useful in tasks such as image-to-image translation and generative modeling.

1. **Group Normalization (GN):** Group Normalization is a technique that divides the channels of a feature map into groups and performs normalization within each group. It aims to address the limitations of BN in scenarios with smaller batch sizes, non-i.i.d. samples, or when the spatial dimensions are not compatible with batch-wise statistics. GN calculates the mean and variance for each group separately and normalizes the activations accordingly.

* **Proper Use**: Group Normalization is effective in situations where the batch size is small or when batch-wise statistics may not be reliable. It is commonly used in scenarios such as object detection, where the spatial dimensions of feature maps are different across samples. GN provides an alternative normalization technique that can be more robust in such cases.

**Q.2) What is mean by hyper parameter enlist various model parameters and Hyperparameters in deep learning?**

* **Hyperparameters** refer to the external configuration settings that determine the architecture and behavior of a neural network model. These settings are set before the training process begins and are not learned from the data itself. Hyperparameters significantly influence how the model is trained and how it generalizes to new, unseen data. Hyperparameters control various aspects of the neural network, such as its structure, optimization algorithm, and regularization techniques. By adjusting these hyperparameters, you can fine-tune the model's performance and improve its ability to learn complex patterns from the data.

1. **Model Parameters:** Model parameters are the internal variables of a neural network that are learned during the training process. They are responsible for capturing the patterns and relationships within the data. These parameters are updated iteratively using optimization algorithms like gradient descent to minimize the difference between the predicted and actual outputs. Examples of model parameters include:

* **Weights:** The synaptic connections between neurons that determine the strength of the signal passed between them.
* **Biases:** The offsets or thresholds added to the weighted inputs of neurons.

1. **Hyperparameters:** Hyperparameters, on the other hand, are external configuration settings that define the architecture and behavior of the neural network. They are set before the training process and influence how the model is trained but are not directly learned from the data. Hyperparameters need to be fine-tuned and experimented with to achieve better model performance. Examples of hyperparameters include:

* **Learning Rate**: Controls the step size taken during gradient descent, affecting the rate at which the model learns.
* **Number of Layers:** Determines the depth of the neural network architecture.
* **Number of Neurons:** Specifies the number of neurons in each layer of the neural network.
* **Activation Functions:** Defines the mathematical functions applied to the output of each neuron.
* **Batch Size:** Determines the number of training examples processed in one forward/backward pass during each training iteration.
* **Dropout Rate:** Controls the regularization technique that randomly drops a fraction of the neurons during training to prevent overfitting.

**Q.3) Explain VGG16net Architecture, With diagram and Trainable params?**

|  |
| --- |
| Description: VGG16 Artitecture |

* The VGG16 architecture is a deep convolutional neural network that achieved state-of-the-art performance on the ImageNet dataset in 2014. It is composed of several convolutional layers with small filter sizes, followed by max pooling layers and fully connected layers. Here is a breakdown of the architecture, along with the number of trainable parameters in each layer:

1. **Input layer:** This layer accepts input images of size 224x224x3 (where 3 is the number of input channels for RGB images). This layer has no trainable parameters.
2. **Convolutional layer 1:** This layer has 64 filters of size 3x3x3, with a stride of 1 pixel and padding of size 1. This layer has approximately 1.7 million trainable parameters.
3. **Convolutional layer 2:** This layer has 64 filters of size 3x3x64, with a stride of 1 pixel and padding of size 1. This layer has approximately 36 million trainable parameters.
4. **Max pooling layer 1:** This layer performs max pooling over non-overlapping regions of size 2x2, with a stride of 2 pixels. This layer has no trainable parameters.
5. **Convolutional layer 3:** This layer has 128 filters of size 3x3x64, with a stride of 1 pixel and padding of size 1. This layer has approximately 73 million trainable parameters.
6. **Convolutional layer 4:** This layer has 128 filters of size 3x3x128, with a stride of 1 pixel and padding of size 1. This layer has approximately 147 million trainable parameters.
7. **Max pooling layer 2:** This layer performs max pooling over non-overlapping regions of size 2x2, with a stride of 2 pixels. This layer has no trainable parameters.
8. **Convolutional layer 5:** This layer has 256 filters of size 3x3x128, with a stride of 1 pixel and padding of size 1. This layer has approximately 295 million trainable parameters.
9. **Convolutional layer 6:** This layer has 256 filters of size 3x3x256, with a stride of 1 pixel and padding of size 1. This layer has approximately 590 million trainable parameters.
10. **Convolutional layer 7:** This layer has 256 filters of size 3x3x256, with a stride of 1 pixel and padding of size 1. This layer has approximately 590 million trainable parameters.
11. **Max pooling layer 3:** This layer performs max pooling over non-overlapping regions of size 2x2, with a stride of 2 pixels. This layer has no trainable parameters.
12. **Convolutional layer 8:** This layer has 512 filters of size 3x3x256, with a stride of 1 pixel and padding of size 1. This layer has approximately 1.2 billion trainable parameters.
13. **Convolutional layer 9:** This layer has 512 filters of size 3x3x512, with a stride of 1 pixel and padding of size 1. This layer has approximately 2.4 billion trainable parameters.
14. **Convolutional layer 10:** This layer has 512 filters of size 3x3x512, with a stride of 1 pixel and padding of size 1. This layer has approximately 2.4 billion trainable parameters.
15. **Max pooling layer 4:** This layer performs max pooling over non-overlapping regions of size 2x2, with a stride of 2 pixels. This layer has no trainable parameters.
16. **Convolutional layer 11:** This layer has 512 filters of size 3x3x512, with a stride of 1 pixel and padding of size 1. This layer has approximately 2.4 billion trainable parameters.
17. **Convolutional layer 12:** This layer has 512 filters of size 3x3x512, with a stride of 1 pixel and padding of size 1. This layer has approximately 2.4 billion trainable parameters.
18. **Convolutional layer 13:** This layer has 512 filters of size 3x3x512, with a stride of 1 pixel and padding of size 1. This layer has approximately 2.4 billion trainable parameters.
19. **Max pooling layer 5:** This layer performs max pooling over non-overlapping regions of size 2x2, with a stride of 2 pixels. This layer has no trainable parameters.
20. **Fully connected layer 1:** This layer has 4096 units. It takes the output of the last convolutional layer, flattens it into a vector, and applies a weight matrix to compute the output of the layer. This layer has approximately 102 million trainable parameters.
21. **Dropout layer 1:** This layer randomly drops out some of the activations from the previous layer during training, to prevent overfitting. This layer has no trainable parameters.
22. **Fully connected layer 2:** This layer has 4096 units. It takes the output of the previous layer and applies another weight matrix to compute the output of the layer. This layer has approximately 16 million trainable parameters.
23. **Dropout layer 2:** This layer randomly drops out some of the activations from the previous layer during training, to prevent overfitting. This layer has no trainable parameters.
24. **Fully connected layer 3:** This layer has 1000 units, one for each class in the ImageNet dataset. It takes the output of the previous layer and applies another weight matrix to compute the output of the layer. This layer has approximately 4 million trainable parameters.
25. **Softmax layer:** This layer takes the output of the previous layer and applies the softmax function to compute the probabilities for each class. This layer has no trainable parameters.

In total, the VGG16 architecture has approximately 138 million trainable parameters. The use of smaller filter sizes (3x3) in the convolutional layers allows for a deeper network with fewer parameters than previous architectures, while still achieving state-of-the-art performance on the ImageNet dataset.

**Q.5) Explain The steps for building image classification model keras with suitable functions?**

* **Here are the steps for building an image classification model using Keras:**

1. **Import required libraries:** The first step is to import the required libraries. In particular, we will be using Keras, which is a high-level neural networks API that runs on top of TensorFlow, and NumPy, which is a library for numerical computing in Python.

|  |
| --- |
| import keras  from keras.models import Sequential  from keras.layers import Dense, Conv2D, MaxPooling2D, Flatten  import numpy as np |

1. **Load the data:** The next step is to load the image data that we want to classify. This can be done using Keras' **‘ImageDataGenerator’** function, which allows us to load images directly from a directory. We can also specify how we want to preprocess the images, such as by scaling their pixel values.

|  |
| --- |
| from keras.preprocessing.image import ImageDataGenerator  # specify data directories  train\_dir = 'path/to/train/data'  val\_dir = 'path/to/validation/data'  test\_dir = 'path/to/test/data'  # create ImageDataGenerator objects for preprocessing data  train\_datagen = ImageDataGenerator(rescale=1./255)  val\_datagen = ImageDataGenerator(rescale=1./255)  test\_datagen = ImageDataGenerator(rescale=1./255)  # load data from directories and apply preprocessing  train\_data = train\_datagen.flow\_from\_directory(train\_dir, target\_size=(224, 224), batch\_size=32)  val\_data = val\_datagen.flow\_from\_directory(val\_dir, target\_size=(224, 224), batch\_size=32)  test\_data = test\_datagen.flow\_from\_directory(test\_dir, target\_size=(224, 224), batch\_size=32) |

1. **Define the model:** The next step is to define the architecture of the neural network. This can be done using the ‘**Sequential’** function in Keras, which allows us to add layers to the network one at a time. We can add convolutional layers, pooling layers, and fully connected layers, among others. We also need to specify the input shape of the images that we are using.

|  |
| --- |
| # define the model  model = Sequential()  model.add(Conv2D(32, (3, 3), activation='relu', input\_shape=(224, 224, 3)))  model.add(MaxPooling2D((2, 2)))  model.add(Conv2D(64, (3, 3), activation='relu'))  model.add(MaxPooling2D((2, 2)))  model.add(Conv2D(128, (3, 3), activation='relu'))  model.add(MaxPooling2D((2, 2)))  model.add(Conv2D(128, (3, 3), activation='relu'))  model.add(MaxPooling2D((2, 2)))  model.add(Flatten())  model.add(Dense(512, activation='relu'))  model.add(Dense(1, activation='sigmoid')) |

1. **Compile the model:** The next step is to compile the model, which involves specifying the loss function, the optimizer, and the evaluation metric that we want to use during training. For a binary classification problem like image classification, we can use binary crossentropy as the loss function and Adam as the optimizer.

|  |
| --- |
| # compile the model  model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy']) |

1. **Train the model:** The next step is to train the model using the training data. This can be done using the **‘fit\_generator’** function in Keras, which allows us to specify the training data, the number of epochs, and the validation data. We can also specify the batch size, which determines the number of samples that are processed at once during training.

|  |
| --- |
| # train the model  history = model.fit\_generator(train\_data, epochs=10, validation\_data=val\_data) |

1. **Evaluate the model:** Once the model has been trained, we can evaluate its performance on the test data using the **‘evaluate\_generator’** function in Keras. This will give us the final accuracy and loss values for the model.

|  |
| --- |
| # evaluate the model  loss, acc = model.evaluate\_generator(test\_data)  print('Test accuracy:', acc) |

1. **Make predictions:** Finally, we can use the trained model to make predictions on new images using the **‘predict’** function in Keras. This will give us the predicted class probabilities for each image.

|  |
| --- |
| # make predictions  predictions = model.predict(test\_data) |

Overall, these are the basic steps for building an image classification model using Keras. Of course, the exact architecture and parameters will depend on the specific problem at hand.

**Q.6) Write a code to load the model sample.h5 in keras and making image predictions with loaded model?**

* Here's an example code to load the model weights from a saved .h5 file and use the model to make predictions on a sample image:

|  |
| --- |
| import numpy as np  import tensorflow as tf  from tensorflow.keras.preprocessing import image  from tensorflow.keras.applications.resnet50 import preprocess\_input, decode\_predictions  from tensorflow.keras.models import load\_model  **# load the saved model**  model = load\_model('sample.h5')  **# load an example image and preprocess it**  img\_path = 'example\_image.jpg'  img = image.load\_img(img\_path, target\_size=(224, 224))  x = image.img\_to\_array(img)  x = np.expand\_dims(x, axis=0)  x = preprocess\_input(x)  **# make a prediction on the image**  preds = model.predict(x)  **# print the predicted class and confidence score**  print('Predicted class:', decode\_predictions(preds, top=1)[0][0][1])  print('Confidence score:', decode\_predictions(preds, top=1)[0][0][2]) |

This code assumes that you have a saved Keras model in the sample.h5 file and an example image in the example\_image.jpg file. The preprocess\_input function is used to preprocess the input image in the same way as the original ResNet50 model was trained. The decode\_predictions function is used to convert the output of the model, which is a probability distribution over the possible classes, into a human-readable format.

**Q.7) Explain in detail training proceedure of CNN?**

* **The training procedure for a Convolutional Neural Network (CNN) involves the following steps:**

1. **Prepare the data:** The first step is to prepare the data for training. This includes dividing the data into training, validation, and test sets, as well as preprocessing the data, such as rescaling the pixel values to be between 0 and 1.
2. **Define the architecture:** The next step is to define the architecture of the CNN. This involves specifying the number of layers, the type of layers, and the hyperparameters for each layer, such as the number of filters in each convolutional layer, the size of the pooling window, the activation function, and the dropout rate.
3. **Compile the model:** Once the architecture is defined, the next step is to compile the model. This involves specifying the loss function, the optimizer, and any performance metrics to be used during training.
4. **Train the model:** After the model is compiled, the next step is to train the model on the training data. This involves feeding the training data through the model and adjusting the weights of the model based on the error between the predicted output and the actual output. During training, the model is also evaluated on the validation set to monitor its performance and prevent overfitting.
5. **Evaluate the model:** Once the model has been trained, the next step is to evaluate its performance on the test set. This involves feeding the test data through the model and measuring its accuracy, as well as any other performance metrics that were specified during compilation.
6. **Fine-tune the model:** After evaluating the model, it may be necessary to fine-tune the architecture or hyperparameters to improve its performance. This involves adjusting the model and repeating the training process until the desired level of performance is achieved.
7. **Use the model for predictions**: Once the model has been trained and evaluated, it can be used to make predictions on new data. This involves feeding the new data through the model and using the predicted output to make decisions or classifications.

Overall, the training procedure for a CNN is similar to that of other types of neural networks, with the main difference being the architecture and hyperparameters specific to CNNs. With careful selection of the architecture and hyperparameters, along with proper training and evaluation, CNNs can achieve state-of-the-art performance on a wide range of image classification tasks.

**Q.8) Explain the advantages of Transfer learning?**

* **Transfer learning** is a technique in deep learning where a pre-trained model is used as the starting point for a new task. The pre-trained model has already learned to recognize useful features from a large dataset, and this knowledge can be leveraged to solve a related task with a smaller dataset. Transfer learning offers several advantages in deep learning:

1. **Reduced training time**: Training a deep neural network from scratch can be time-consuming and computationally expensive, especially when working with large datasets. Transfer learning allows us to start with a pre-trained model and fine-tune it for the new task. This can reduce the number of training epochs required to achieve high accuracy and save significant training time.
2. **Improved generalization:** Pre-trained models have already learned to recognize features that are useful for a particular task. By leveraging this knowledge, transfer learning can help to improve the generalization performance of a model on a related task. This is especially useful when the new task has a smaller dataset, as the pre-trained model can prevent overfitting and improve the model's ability to generalize to new data.
3. **Higher accuracy:** Transfer learning can lead to higher accuracy on a new task, particularly when the pre-trained model was trained on a related task or dataset. The pre-trained model has already learned to recognize important features, which can be fine-tuned for the new task. This can lead to better accuracy compared to training a model from scratch.
4. **Reduced data requirements:** Transfer learning can be useful when the new task has limited training data. By starting with a pre-trained model, the model can leverage the knowledge that has been learned from a larger dataset, even when the new task has only a small amount of data.
5. **Access to state-of-the-art models:** Transfer learning allows access to state-of-the-art models that have been trained on large datasets. These models can be difficult and time-consuming to train from scratch, but with transfer learning, they can be fine-tuned for a specific task with a smaller dataset.

**Q.9) Explain procedure of feature extraction using pretrained network and training classifier head? Consider number of classes as 5?**

* The procedure for feature extraction using a pre-trained network and training a classifier head involves the following steps:
* **Import the pre-trained network:** Choose a pre-trained network that is suitable for the task at hand and import it into your deep learning framework. In this example, we will use the ResNet50 pre-trained network.
* **Freeze the pre-trained network:** Freeze all the layers of the pre-trained network to prevent their weights from being updated during training.
* **Add a classifier head:** Add a new classifier head to the pre-trained network. The classifier head is a set of fully connected layers that are responsible for producing the output predictions. In this example, we will add a classifier head with 5 output neurons, one for each class.
* **Prepare the dataset:** Prepare the dataset by splitting it into training and validation sets. In this example, we will assume that we have 500 training images and 100 validation images for each of the 5 classes.
* **Feature extraction:** Use the pre-trained network to extract features from the training and validation images. To do this, pass the images through the pre-trained network and record the output from one of the final layers. This output can be considered as the features for each image.
* **Train the classifier head:** Use the extracted features to train the classifier head. To do this, pass the extracted features through the classifier head and use a suitable loss function (such as cross-entropy loss) to update the weights of the classifier head. Train the classifier head using the training set and evaluate its performance on the validation set.
* **Fine-tuning:** After training the classifier head, fine-tune the pre-trained network by unfreezing some of its layers and retraining the model using a smaller learning rate. This can help to improve the accuracy of the model.
* **Evaluation:** Finally, evaluate the performance of the model on a test set to see how well it generalizes to new data.

The procedure for feature extraction using a pre-trained network and training a classifier head involves using the pre-trained network to extract features from the data, training a new classifier head on top of the extracted features, and fine-tuning the pre-trained network to improve the accuracy of the model. This technique can be useful when working with limited data and can help to achieve good results with less training time.

**Q.10) Explain the procedure of fine tuning pretrained network using transfer learning. Consider number of classes as 5?**

* The procedure for fine-tuning a pre-trained network using transfer learning involves the following steps:

1. **Import the pre-trained network**: Choose a pre-trained network that is suitable for the task at hand and import it into your deep learning framework. In this example, we will use the ResNet50 pre-trained network.
2. **Modify the classifier head:** Replace the existing classifier head of the pre-trained network with a new one. The new classifier head should have 5 output neurons, one for each class in our example.
3. **Freeze some layers:** Freeze some of the layers in the pre-trained network to prevent their weights from being updated during training. Typically, the first few layers are frozen and the later layers are left trainable.
4. **Prepare the dataset:** Prepare the dataset by splitting it into training and validation sets. In this example, we will assume that we have 500 training images and 100 validation images for each of the 5 classes.
5. **Fine-tune the network:** Fine-tune the pre-trained network by training the entire model on the training set and updating the weights of the trainable layers. Use a suitable loss function (such as cross-entropy loss) to update the weights of the model.
6. **Evaluate the model:** Evaluate the performance of the fine-tuned model on the validation set to see how well it generalizes to new data.
7. **Adjust the hyperparameters:** Adjust the hyperparameters of the model, such as the learning rate and batch size, to optimize the performance on the validation set.
8. **Test the model:** Finally, test the performance of the model on a test set to see how well it performs on new, unseen data.

In summary, the procedure for fine-tuning a pre-trained network using transfer learning involves modifying the existing classifier head, freezing some layers, and fine-tuning the model on the training data. This technique can be useful when working with limited data and can help to achieve good results with less training time compared to training a model from scratch.