**NN & DeepLearning : Image Classification with CNN\_ICP8**

**Name**: Maddikunta Dakshyani

#**700**: 700666204

* **Task 1:** Tune hyperparameter and make necessary addition to the baseline model to improve validation accuracy and reduce validation loss.
  + In the provided code snippets, the primary hyperparameter that has been tuned to potentially improve validation accuracy and reduce validation loss is the configuration of the Stochastic Gradient Descent (SGD) optimizer.
  + Specifically, the key parameters that affect the optimizer's behavior and, consequently, the model's training dynamics are:

**1. Learning Rate (learning\_rate):**

The learning rate determines the step size taken during parameter updates in the direction opposite to the gradient.

Adjusting the learning rate can have a significant impact on the convergence speed and the quality of the final solution.

**2. Decay Rate (decay):**

The decay rate controls the amount by which the learning rate decreases over epochs.

It helps in annealing the learning rate, typically reducing it as training progresses.

A smaller decay rate implies slower decay, keeping the learning rate relatively higher for longer during training.

* + By adjusting these parameters, one can influence how the model learns from the training data, potentially leading to improved validation accuracy and reduced validation loss.
  + In the provided code snippets, the second code snippet fixes the learning rate (learning\_rate=0.01) and decay rate (decay=1e-6). This fixed configuration might not be as effective in adapting to changing training dynamics compared to the adaptive decay strategy used in the first code snippet.
  + Therefore, if the aim is to improve validation accuracy and reduce validation loss, tuning these hyperparameters, particularly the learning rate and decay rate, while monitoring the model's performance on the validation set, could lead to better results.
* **Task 2:** Provide logical description of which steps lead to improved response and what was its impact on architecture behavior.

1. Differences in Optimizer Initialization:

First Code:

python

sgd = SGD(lr=learning\_rate, momentum=0.9, decay=decay\_rate, nesterov=False)

Second Code:

python

sgd = SGD(learning\_rate=0.01, momentum=0.9, decay=1e-6)

In the first code, the learning rate and decay rate are calculated based on predefined variables (learning\_rate and decay\_rate). In the second code, these values are directly set as parameters during the optimizer initialization.

2. Impact on Learning Rate and Decay:

In the first code, the learning rate and decay rate are calculated based on provided values (learning\_rate and epochs). The decay rate decreases the learning rate over epochs, which could be more adaptive to the training process.

In the second code, the learning rate and decay rate are fixed (learning\_rate=0.01 and decay=1e-6). This means the learning rate starts at 0.01 and decreases by a fixed amount (1e-6) over each epoch.

3. Impact on Model Behavior:

First Code:

The learning rate starts at a higher value (learning\_rate) and decreases gradually over epochs (decay\_rate).

This approach might lead to a more adaptive learning process where the model starts with a higher learning rate to make larger updates to its parameters in the beginning and then fine-tunes them as it converges.

The decay rate ensures that the learning rate decreases steadily over epochs, potentially helping the model to converge to a better solution over time.

Second Code:

The learning rate starts at a fixed value (0.01) and decreases by a fixed amount (1e-6) over each epoch.

This approach provides a constant learning rate decay throughout training.

While the fixed learning rate decay might simplify the training process, it might not adapt as well to the training dynamics compared to the adaptive decay in the first code.

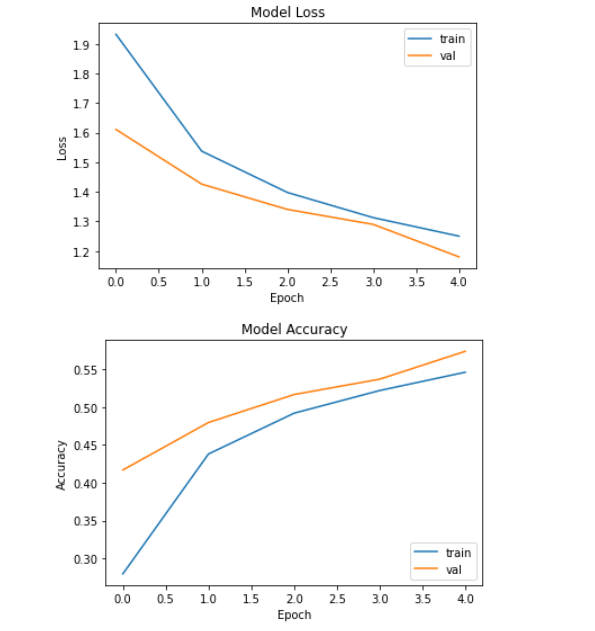
4. Overall Impact:

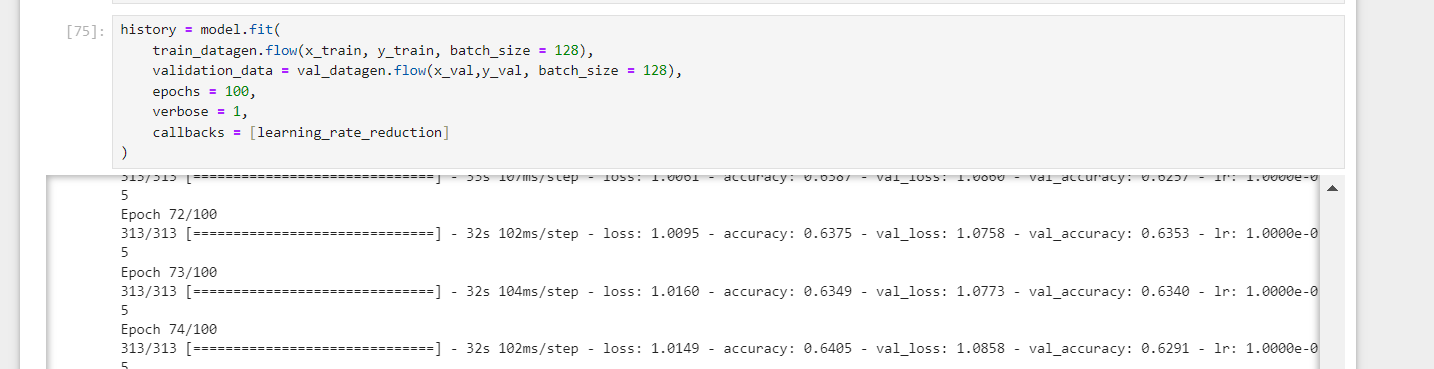
The first code allows for more adaptability in the learning rate, potentially leading to better convergence and performance, especially if the training dynamics change significantly over epochs.

The second code simplifies the training process by fixing the learning rate and decay, but it might not be as effective in adapting to changing training dynamics.

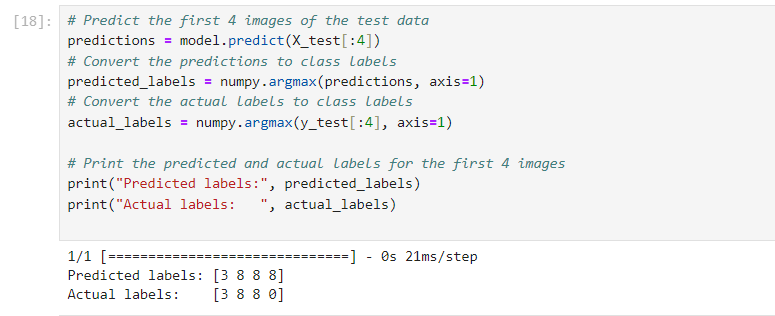
In summary, the choice between the two approaches depends on factors such as the complexity of the dataset, the dynamics of the training process, and the desired trade-off between simplicity and adaptability in the learning process.

* **Task 3:** Create at least two more visualizations using matplotlib (Other than provided in the source file).



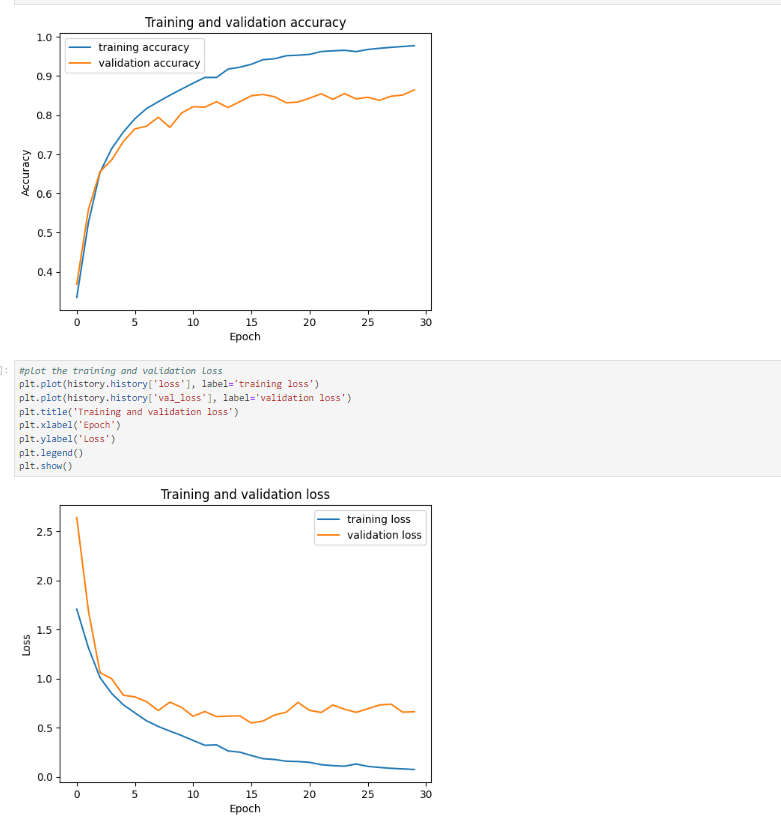
* **Task:** Use dataset of your own choice and implement baseline models provided

We implement the base model and train the dataset with increased epochs i.e. 100 in VGG19 model and then saved the improved model and used it for prediction on our testing dataset.



** Task:** Provide plot of confusion matric.

**Task:** Provide Training and testing Loss and accuracy plots in one plot using subplot command and history object.

****

**Task:** Provide logical description of which steps lead to improved response for new dataset when compared with baseline model and enhance architecture and what was its impact on architecture behavior.

The modifications made to the baseline VGG16-like model include the addition of Batch Normalization layers after each convolutional layer and Dropout layers after each max-pooling layer. Here's a logical description of how these modifications can potentially lead to improved performance for a new dataset:

**1. Batch Normalization:**

**Improved Convergence:** Batch Normalization normalizes the activations of each layer, reducing internal covariate shift. This stabilizes and speeds up the training process by ensuring that the inputs to each layer are consistently scaled and centered. As a result, the network can converge faster during training.

**Reduced Sensitivity to Initialization:** Batch Normalization makes the network less sensitive to the choice of initialization parameters. This allows for more aggressive initialization strategies, potentially leading to better performance and faster convergence.

**Regularization Effect:** Batch Normalization acts as a form of regularization by adding noise to the activations. This noise helps prevent overfitting and improves the generalization performance of the model.

**2. Dropout:**

**Regularization:** Dropout layers randomly drop a fraction of the units' outputs during training, effectively making the network more robust and less prone to overfitting. By preventing units from co-adapting too much, Dropout encourages the network to learn more robust features.

**Reduced Overfitting:** Dropout helps prevent the network from memorizing the training data by introducing noise during training. This forces the network to learn more generalizable features that are applicable to unseen data, thus reducing overfitting.

**Ensemble Effect:** Dropout can be interpreted as training multiple subnetworks with shared parameters. During inference, these subnetworks are combined, effectively creating an ensemble of models. This ensemble approach often leads to improved generalization performance.

**3. Impact on Architecture Behavior:**

**Faster Convergence:** Batch Normalization stabilizes and speeds up the training process, leading to faster convergence to a good solution.

**Reduced Overfitting:** Dropout layers act as a regularization technique, preventing the network from overfitting to the training data and improving its ability to generalize to unseen data.

**Improved Generalization:** The combination of Batch Normalization and Dropout helps the model learn more robust features and reduces its sensitivity to noisy input, ultimately improving its generalization performance on unseen data.

Overall, the addition of Batch Normalization and Dropout layers enhances the baseline VGG16-like model by improving convergence speed, reducing overfitting, and enhancing its ability to generalize to new datasets.

