Name: Afsah ur Rehman
Roll no.: bscs22119

Assignmnet 03

For each task we will have 3 diff versions and each version will have diff different hyperparameter values,

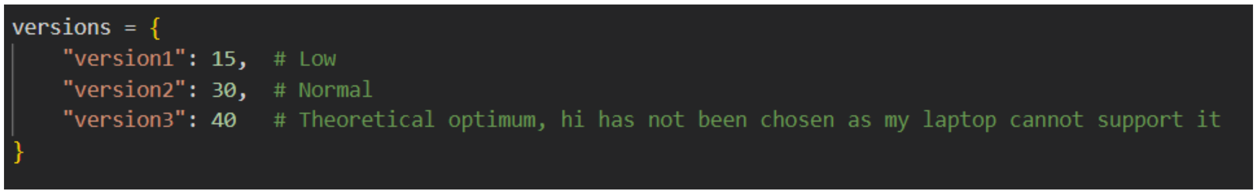
- Low values
- Normal values
- Optimistic values using laws and theorems for hyperparas value calculation
  No high values as my laptop can't support them.

Also I have mattresses that will measure which is the best version, And also then I have mattresses that will compare the best version of each technique with all the best version of the techniques,
P.S: I thought cpu utilization will be a good thing to move my ear but it wasn't on a scale as I was doing other stuff also on my laptop, so its kinda useless for now.

Task 1

Context:

1. Mechanism
2. version three law
3. Hyperparameter their impacts
4. Visualizations and outputs
5. Things asked in documentation



```
versions = {
    "version1": 15,  # Low
    "version2": 30,  # Normal
    "version3": 40   # Theoretical optimum, hi has not been chosen as my laptop cannot support it
}
```

Mechanism

- **Step 1: Load the CIFAR-10 Data**
- **Step 2: Preprocess the Images**

  Normalization: Convert pixel values from 0–255 to 0–1. So that ensures all pixels are on a common scale, and we wil do it by dividing each pixel value with 255.

$$X\text{-}normalized = X\text{-}original / 255$$

Reshape: Flatten each image. Flattening turns images into vectors so we can perform mathematical operations and we will do this by *reshape()* function in py.

- **Step 3: Apply PCA on the Training Set for Each Class**
  1. Compute the Mean (μ)
  2. Center the Data to the origin by
  3. *data_centered = data_i - μ*
  4. Compute the Covariance Matrix (Sigma: Σ), this will capture the relation b/w the features
  5. $\Sigma = \sum_{i}^{N}(data_{centered}) \ * \ (data_{centered})^{T}$

- **Step 4: Eigen-Decomposition:** bec Eigenvectors (principal components) reveal the most significant patterns

$$\Sigma v = \lambda v$$

  1. **v:** Eigenvector (direction of maximum variance)
  2. **λ:** Eigenvalue (amount of variance in that direction)
- **Step 5: Select Top k Eigenvectors,** These are the principal components used for reconstruction. Using only the top k components reduces dimensionality while keeping most variance.
- **Step 6 Reconstruct an Image:**
  To approximate the original image using a reduced number of components.

$$\hat{x} = \mu + \sum_{i=1}^{k} a_i v_i$$

  1. $\hat{x}$ is the Constructed image
  2. a_i = Coefficient calculated as $v_i^T(x - \mu)$
  3. v_i is the i-th eigenvector
- **Step 7: Compute Reconstruction Error on the Test Set**

  For each test image, reconstruct it using each class's PCA model and calculate the error. The error tells us how well each class's PCA model represents the image. And we will use MSE for this,

MSE = I/N summation of N from i (x_i - $\hat{x}$_i)

- **Step 8: Evaluate Classification Accuracy** by simple accuracy formula

Acc = correct / total * 100%

- **Step 9: Visualize Reconstructed Images**
- **Step 10: save the Different model versions**
  For saving I have use 2 things:
  1. **Mean File ("class_X_mean.npy"):**
     This file stores the mean vector (μ) for all training images in that class. In PCA, we subtract the mean from each image to center the data before computing the covariance matrix.
  2. **Principal Components File ("class_X_pcs.npy"):**
     This file stores the principal components (eigenvectors) for that class. These vectors capture the directions in which the data varies the most, and we use them for projecting the data and later reconstructing the image.

Version 3 law:

We are using the Kaiser criterion. In simple terms, it means we only keep those principal components whose eigenvalues are greater than 1. This rule helps us decide to keep only the components that explain more information than one of the original features.

Hyperparamters with their impacts

The only hyperparameter here is key which is the number of principle components that we are selecting from Eigenvalues

Lower Value of k (Fewer Principal Components):

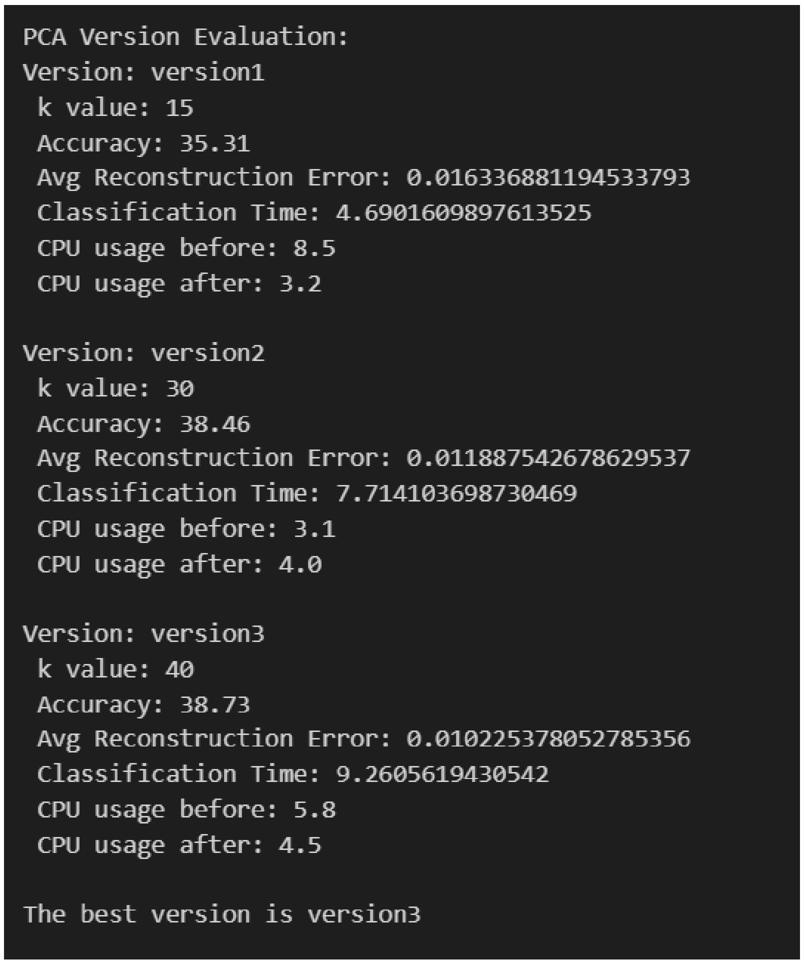
- **Pros:** Faster computation and lower memory usage, which is beneficial on systems without a dedicated GPU.
- **Cons:**May lose important information and variance in the data. And can result in poor image reconstruction quality and lower classification accuracy.

Higher Value of k (More Principal Components):

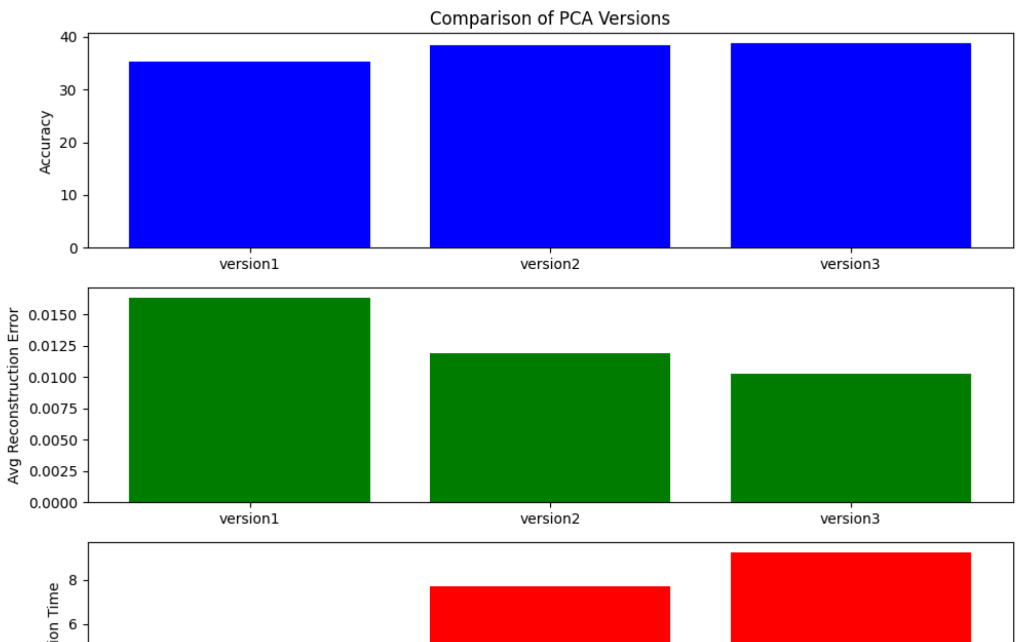
- **Pros:** Retains more information and variance, leading to better image reconstruction and potentially higher accuracy.
- **Cons:**Increases computational load and memory requirements, which can be challenging on non-GPU systems.May incorporate redundant or noisy components that do not contribute meaningfully, possibly degrading overall performance.

Outputs and visualizations:

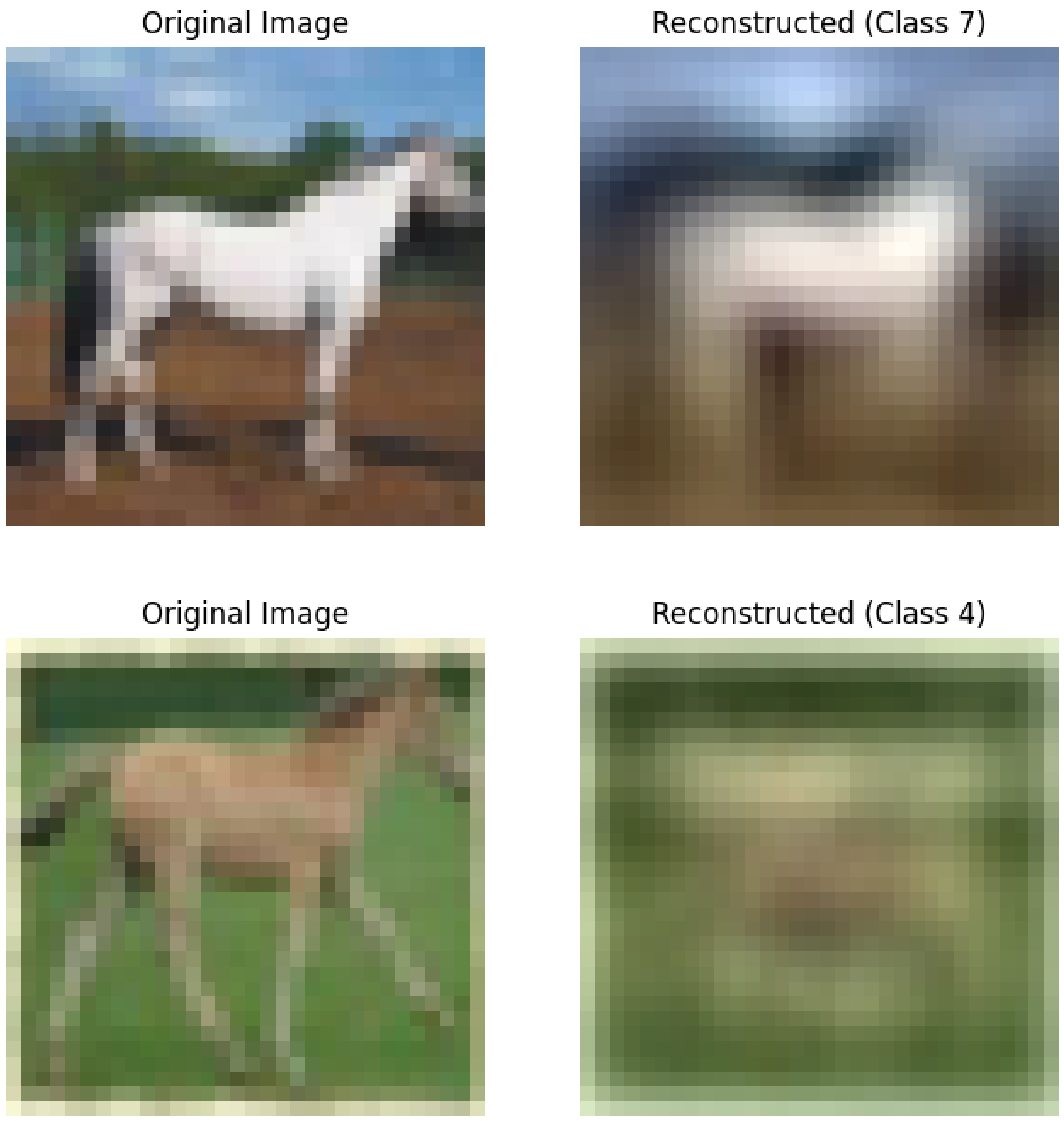
Best version is version 3



```
PCA Version Evaluation:
Version: version1
 k value: 15
 Accuracy: 35.31
 Avg Reconstruction Error: 0.016336881194533793
 Classification Time: 4.6901609897613525
 CPU usage before: 8.5
 CPU usage after: 3.2

Version: version2
 k value: 30
 Accuracy: 38.46
 Avg Reconstruction Error: 0.011887542678629537
 Classification Time: 7.714103698730469
 CPU usage before: 3.1
 CPU usage after: 4.0

Version: version3
 k value: 40
 Accuracy: 38.73
 Avg Reconstruction Error: 0.010225378052785356
 Classification Time: 9.2605619430542
 CPU usage before: 5.8
 CPU usage after: 4.5

The best version is version3
```



Reconstructed images for veriosn 3



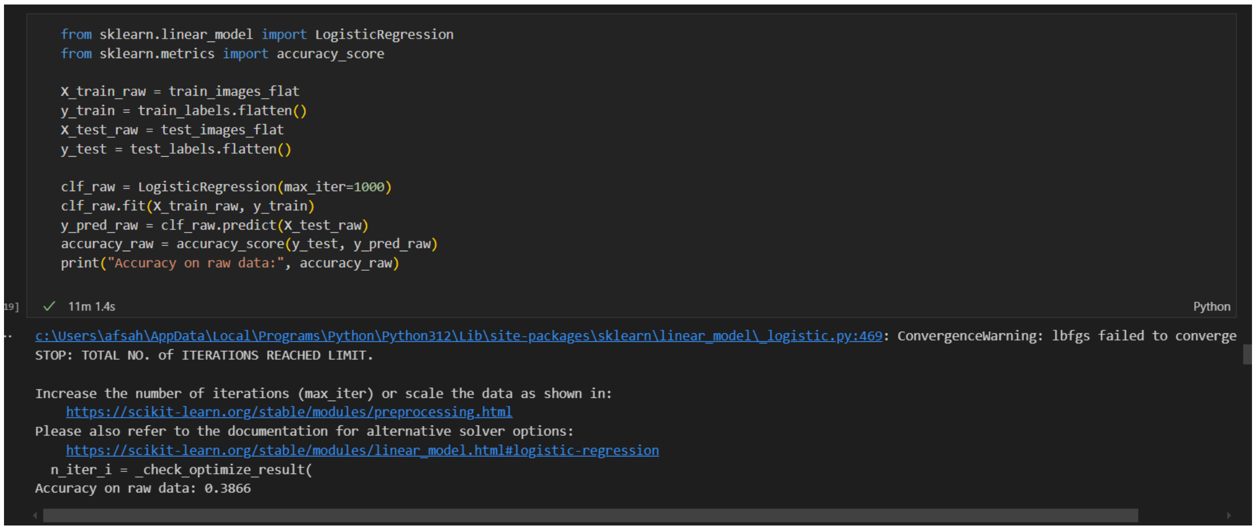
Things asked in doc

Discussion on the impact of dimensionality reduction on classification.

High-Dimensional Data (Raw Features):

Using all original features can make training slow and prone to overfitting because of noise and redundant information.

Example: A classifier trained on raw data might struggle to generalize well.



```python
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

X_train_raw = train_images_flat
y_train = train_labels.flatten()
X_test_raw = test_images_flat
y_test = test_labels.flatten()

clf_raw = LogisticRegression(max_iter=1000)
clf_raw.fit(X_train_raw, y_train)
y_pred_raw = clf_raw.predict(X_test_raw)
accuracy_raw = accuracy_score(y_test, y_pred_raw)
print("Accuracy on raw data:", accuracy_raw)
```

✓ 11m 1.4s Python

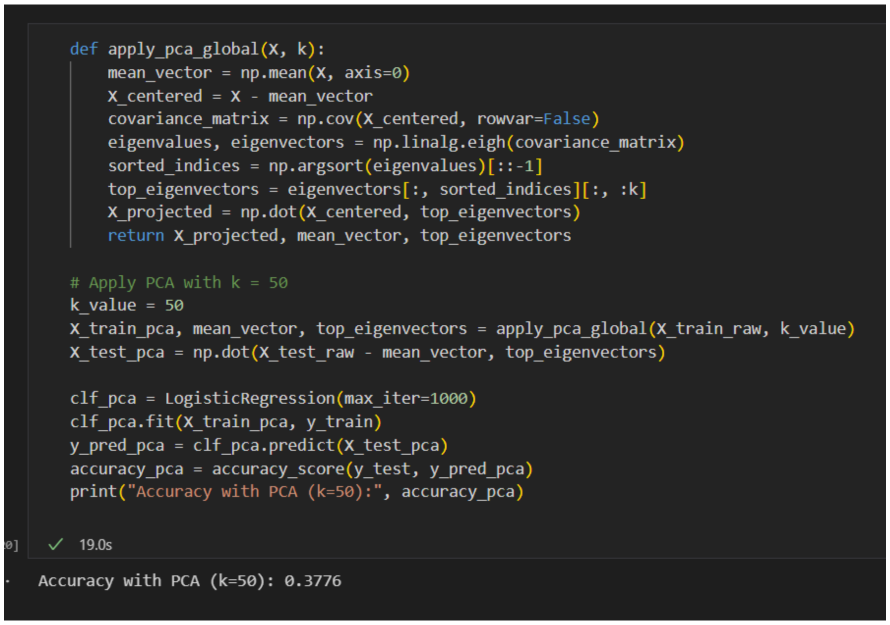
```
c:\Users\afsah\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\linear_model\_logistic.py:469: ConvergenceWarning: lbfgs failed to converge
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
Accuracy on raw data: 0.3866
```

As we can see Daddy took around eleven minutes to complete

Dimensionality Reduction with PCA

PCA reduces the number of features by keeping only the most informative components.This makes the classifier faster and can improve performance by filtering out noise.

Example: Reducing the feature space to 50 components may lead to higher accuracy and faster training. This took around eleven seconds and we are getting about the same accuracy



```python
def apply_pca_global(X, k):
    mean_vector = np.mean(X, axis=0)
    X_centered = X - mean_vector
    covariance_matrix = np.cov(X_centered, rowvar=False)
    eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)
    sorted_indices = np.argsort(eigenvalues)[::-1]
    top_eigenvectors = eigenvectors[:, sorted_indices][:, :k]
    X_projected = np.dot(X_centered, top_eigenvectors)
    return X_projected, mean_vector, top_eigenvectors

# Apply PCA with k = 50
k_value = 50
X_train_pca, mean_vector, top_eigenvectors = apply_pca_global(X_train_raw, k_value)
X_test_pca = np.dot(X_test_raw - mean_vector, top_eigenvectors)

clf_pca = LogisticRegression(max_iter=1000)
clf_pca.fit(X_train_pca, y_train)
y_pred_pca = clf_pca.predict(X_test_pca)
accuracy_pca = accuracy_score(y_test, y_pred_pca)
print("Accuracy with PCA (k=50):", accuracy_pca)
```

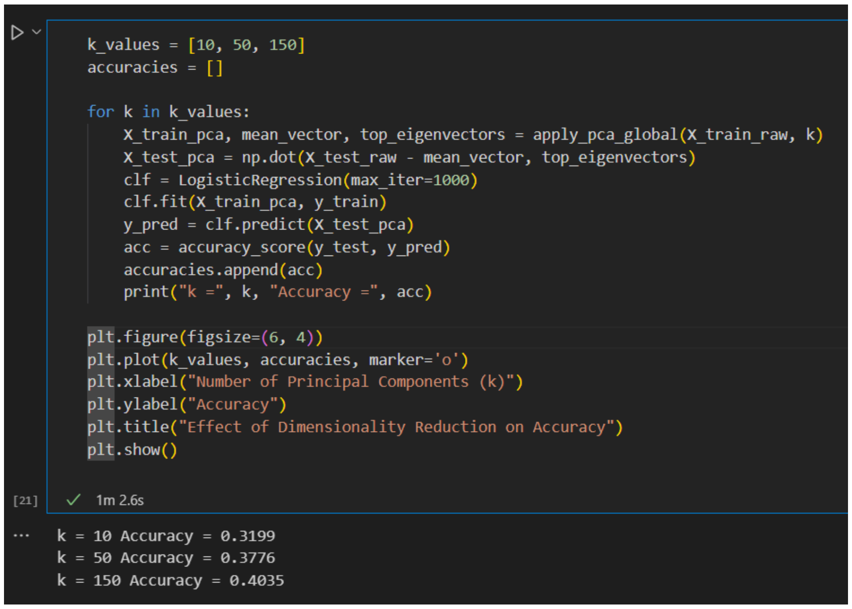
```
✓ 19.0s
Accuracy with PCA (k=50): 0.3776
```

Trade-off in Choosing k (Number of Principal Components):

- **Too Low k:**
  - May lose important information, resulting in poorer accuracy.
- **Too High k:**
  - May include redundant or noisy features, reducing the benefits of PCA and slowing down the classifier.

Example: Testing different k values (such as 10, 50, and 150) shows the trade-off between dimensionality reduction and classification accuracy

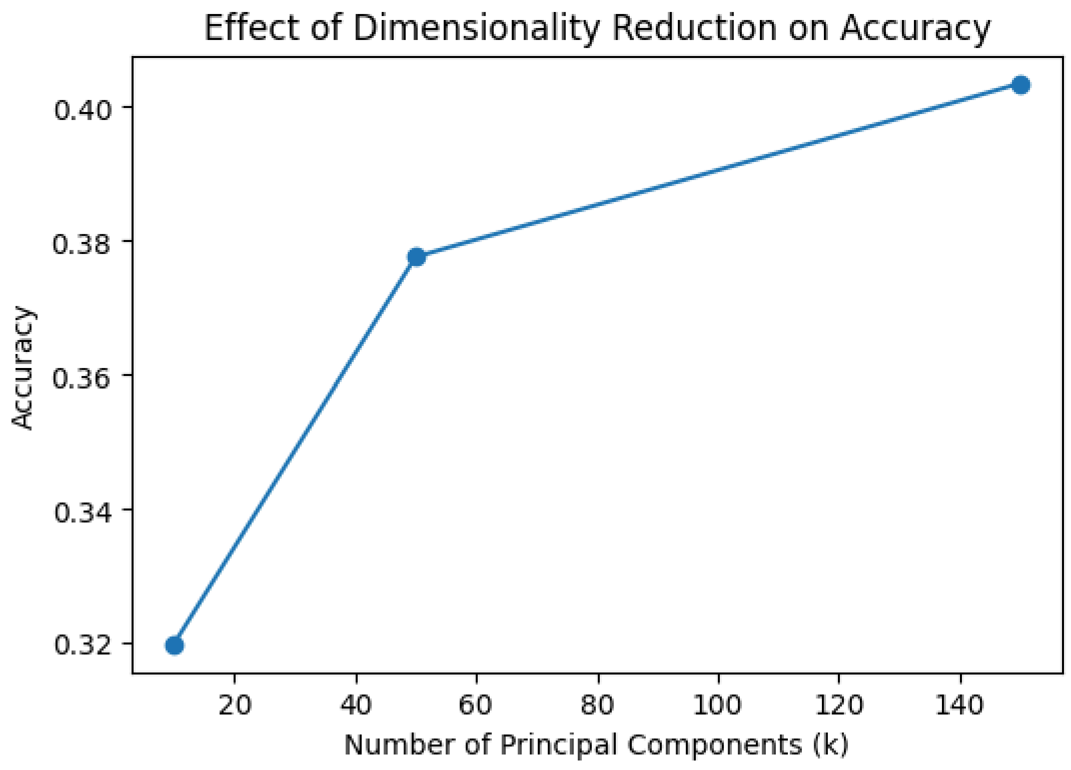
This code only took around one minute and twenty two seconds to complete, And we can see that the different accuracy on different ks



```
k_values = [10, 50, 150]
accuracies = []

for k in k_values:
    X_train_pca, mean_vector, top_eigenvectors = apply_pca_global(X_train_raw, k)
    X_test_pca = np.dot(X_test_raw - mean_vector, top_eigenvectors)
    clf = LogisticRegression(max_iter=1000)
    clf.fit(X_train_pca, y_train)
    y_pred = clf.predict(X_test_pca)
    acc = accuracy_score(y_test, y_pred)
    accuracies.append(acc)
    print("k =", k, "Accuracy =", acc)

plt.figure(figsize=(6, 4))
plt.plot(k_values, accuracies, marker='o')
plt.xlabel("Number of Principal Components (k)")
plt.ylabel("Accuracy")
plt.title("Effect of Dimensionality Reduction on Accuracy")
plt.show()
```

[21] ✓ 1m 2.6s

```
k = 10 Accuracy = 0.3199
k = 50 Accuracy = 0.3776
k = 150 Accuracy = 0.4035
```



Task 2

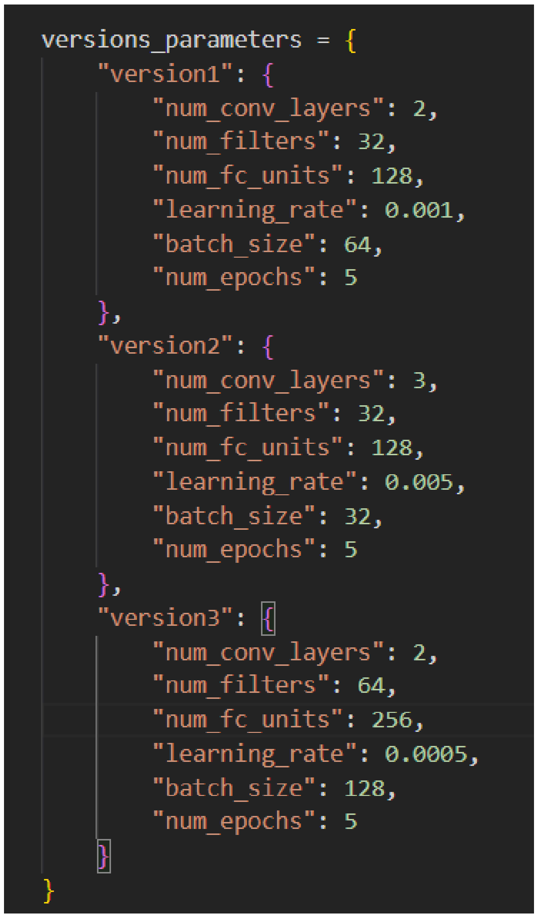
Working

1. **Load and Preprocess Data:**
   Simple loading , idk what to write here
2. **Build the Neural Network:**
   Created a simple convolutional neural network (CNN) using PyTorch. And defined a multi-layer architecture that includes convolutional layers, ReLU activation, pooling layers, and fully connected layers.
3. **Train the Network:**
   Use an optimizer like Adam (or SGD) and the cross-entropy loss function to train the model on the training set. Adjust hyperparameters such as learning rate, number of layers, and batch size to create different model versions.
4. **Evaluate the Model:**
   Test the trained network on the test set and calculate the classification accuracy. Record metrics like training time, loss curves, and accuracy for each version.

Hyperparameter

1. **Learning Rate:**
   The learning rate controls how much the model's weights are updated during each training step. A higher learning rate (for example, 0.005) can lead to faster training but might cause the optimizer to overshoot the optimal values, resulting in unstable or poor convergence. In contrast, a lower learning rate (for example, 0.0005) provides more gradual updates, which can lead to more stable convergence but may slow down the training process.
2. **Number of Convolutional Layers:**
   This parameter defines the depth of the network. A network with fewer convolutional layers (such as 2 layers) is simpler and trains faster, but it might not capture all the complex features in the data. Increasing the number of layers (to 3 or more) can help the network learn more intricate patterns, potentially improving accuracy, but it also increases the computational load and the risk of overfitting if not managed correctly.
3. **Batch Size:**
   Batch size determines how many samples are processed before the model's weights are updated. A smaller batch size (like 32) means more frequent weight updates, which can help the model generalize better but may also lead to noisy updates and slower training. A larger batch size (like 128) makes training faster by reducing the number of updates per epoch; however, it might reduce the stochasticity that can help the model escape local minima and could lead to poorer generalization.
4. **Optimizer and Activation Functions:**
   For this task, we use the Adam optimizer, which adapts the learning rate for each

parameter, making it more efficient for complex networks. Adam is generally robust and works well with most network architectures. The activation functions used are typically ReLU (Rectified Linear Unit), which introduces non-linearity into the network and helps mitigate issues like the vanishing gradient problem. ReLU is simple to compute and generally accelerates convergence compared to other activation functions.



```
versions_parameters = {
    "version1": {
        "num_conv_layers": 2,
        "num_filters": 32,
        "num_fc_units": 128,
        "learning_rate": 0.001,
        "batch_size": 64,
        "num_epochs": 5
    },
    "version2": {
        "num_conv_layers": 3,
        "num_filters": 32,
        "num_fc_units": 128,
        "learning_rate": 0.005,
        "batch_size": 32,
        "num_epochs": 5
    },
    "version3": {
        "num_conv_layers": 2,
        "num_filters": 64,
        "num_fc_units": 256,
        "learning_rate": 0.0005,
        "batch_size": 128,
        "num_epochs": 5
    }
}
```

Version 3: Theoretical Guidelines for Hyperparameters

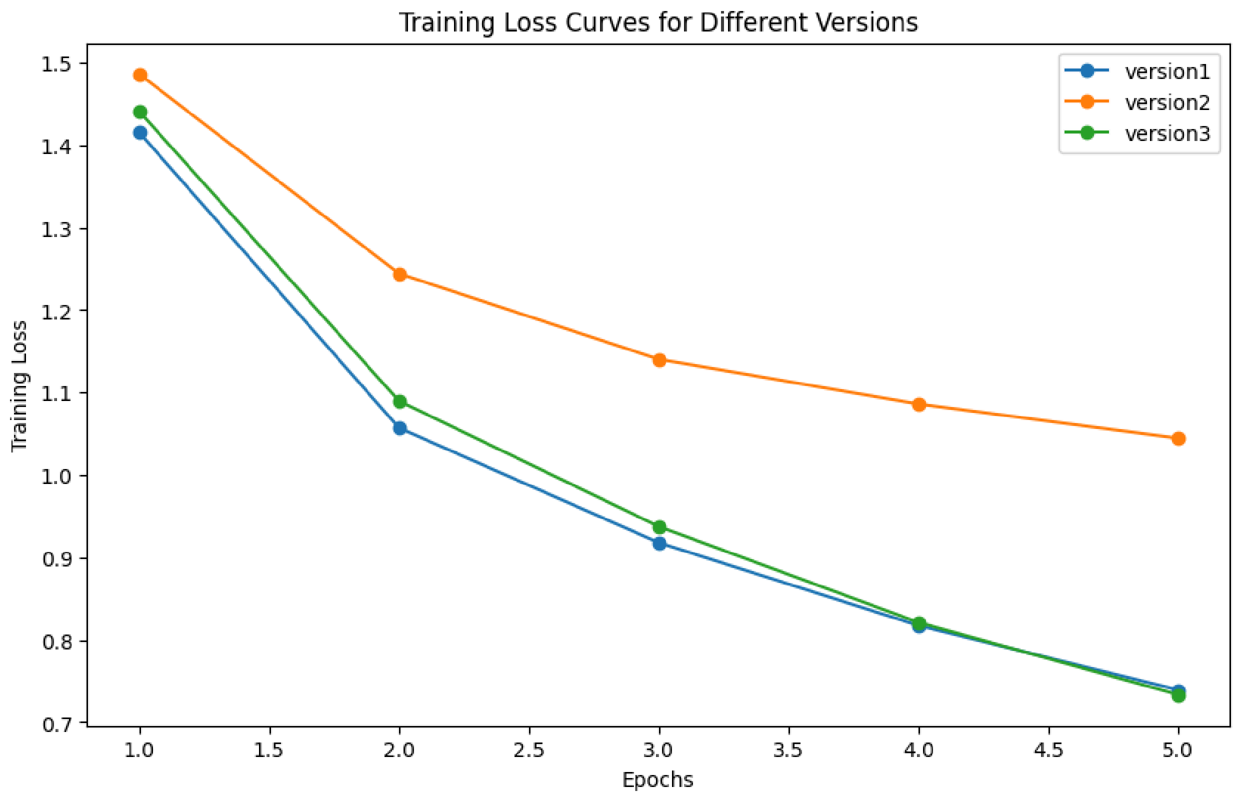
In Version 3, we choose hyperparameter values based on established theories and best practices:

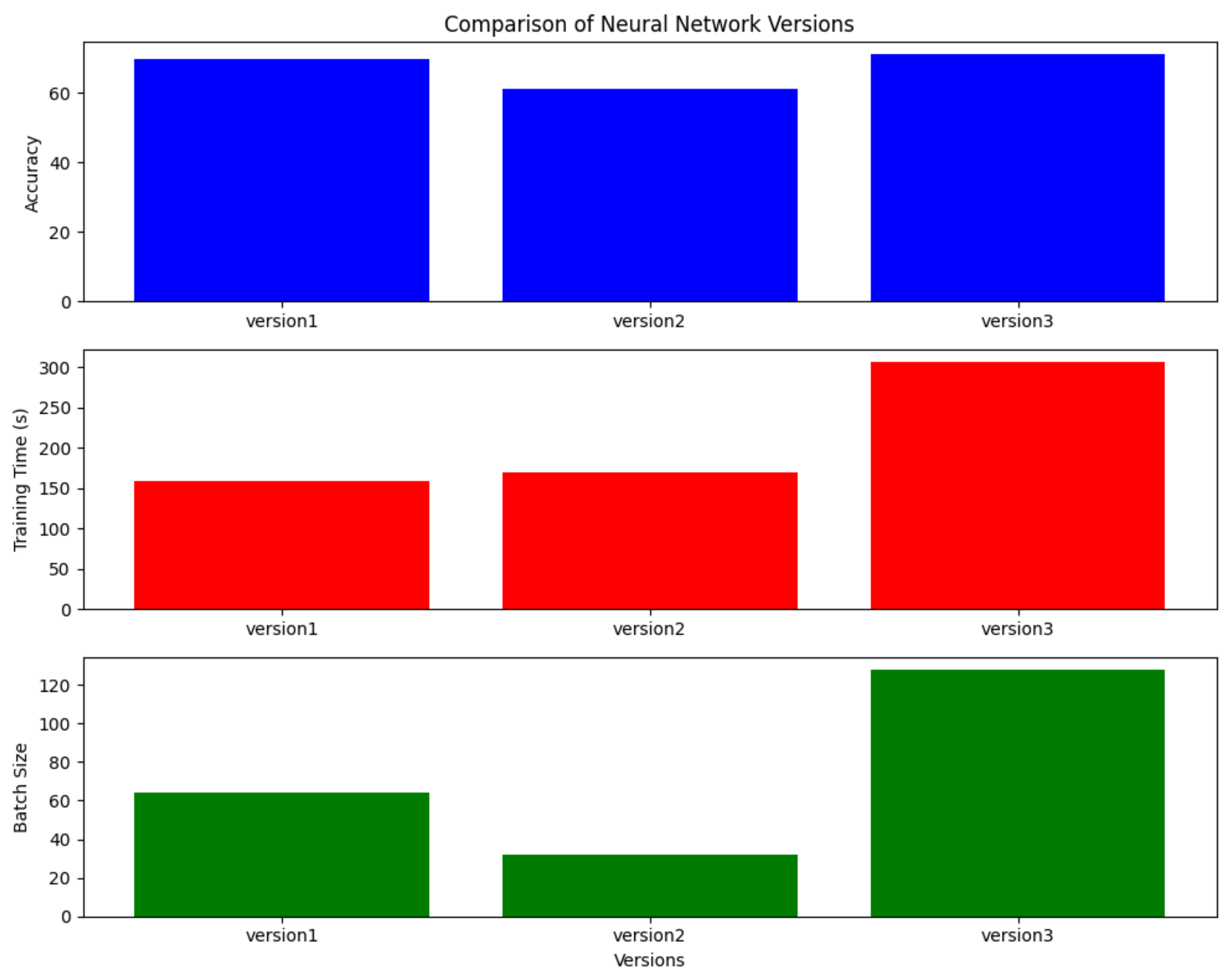
- **Learning Rate:**
  We select a lower learning rate (such as 0.0005) because theory suggests that lower rates can lead to smoother, more stable convergence. This is especially important for deeper or more complex networks to avoid oscillations during training.

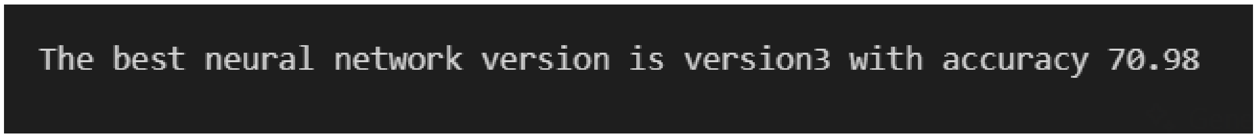
- **Network Depth and Filter Size:**
  We increase the number of filters (for example, using 64 filters instead of 32) and possibly adjust the number of fully connected units (e.g., 256 units). The idea is that with more filters and units, the network can capture more detailed features. However, this is

balanced with the fact that deeper networks require more computational resources. Theory suggests that increasing the model capacity should be done cautiously to avoid overfitting and excessive resource usage.

- **Batch Size:**
  In Version 3, a larger batch size (such as 128) is chosen based on the principle that while larger batches speed up training by taking advantage of parallel processing, they might also smooth out the gradient updates too much. The trade-off here is that with sufficient data and a careful learning rate, a larger batch can be beneficial for faster convergence on a stable system.

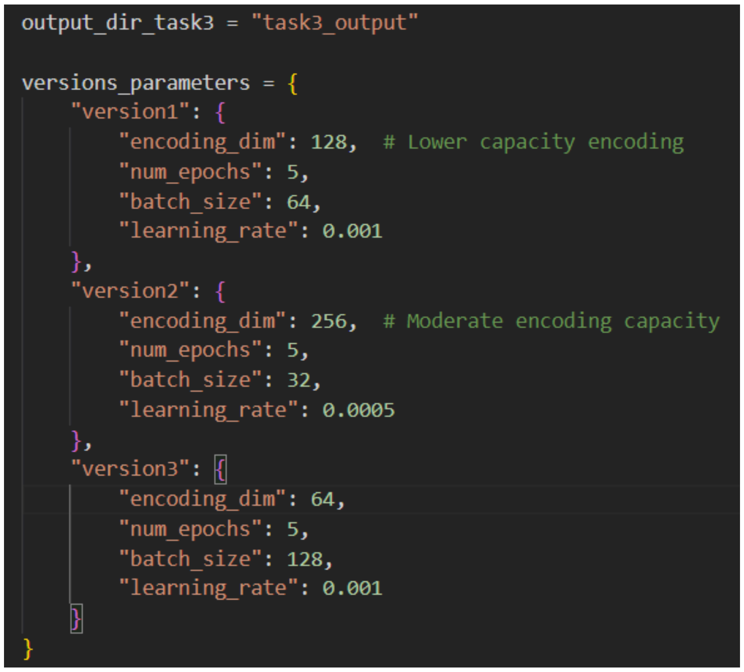






```
The best neural network version is version3 with accuracy 70.98
```

TASK 3



```python
output_dir_task3 = "task3_output"

versions_parameters = {
    "version1": {
        "encoding_dim": 128,  # Lower capacity encoding
        "num_epochs": 5,
        "batch_size": 64,
        "learning_rate": 0.001
    },
    "version2": {
        "encoding_dim": 256,  # Moderate encoding capacity
        "num_epochs": 5,
        "batch_size": 32,
        "learning_rate": 0.0005
    },
    "version3": {
        "encoding_dim": 64,
        "num_epochs": 5,
        "batch_size": 128,
        "learning_rate": 0.001
    }
}
```

Working

1. **Data Preparation:**
    - Load CIFAR-10
    - Apply transforms to convert images to tensors and normalize them (range [-1, 1]).
    - Group training samples by class.
2. **Autoencoder Design and Training:**
    - Define a fully connected autoencoder that flattens input images (3072 features), encodes them to a lower-dimensional space, and then decodes them back to the original size.
    - Train a separate autoencoder for each class using the training data from that class.
    - Use Mean Squared Error (MSE) as the loss function and Adam optimizer for training.
3. **Classification and Evaluation:**
    - For each test image, use each class-specific autoencoder to reconstruct the image and calculate its MSE.
    - Assign the label of the autoencoder with the smallest reconstruction error.
    - Evaluate overall accuracy and visualize the original and reconstructed images side by side.

2. Version 3 Details

- **Version 3 Hyperparameters:**
    - **Encoding Dimension:** 64
    - **Batch Size:** 128
    - **Learning Rate:** 0.001
    - **Number of Epochs:** 5

- **Theoretical Basis:**
    - In Version 3, the chosen encoding dimension is based on theoretical guidelines that suggest a lower-dimensional representation (when sufficiently low) can capture the most critical features while reducing noise.
    - This version uses the principle of retaining only the most significant features in the data, which aligns with ideas from information theory about reducing redundancy.
    - The selected batch size and learning rate are set to balance training stability and speed, ensuring that the network converges without overfitting.

3. Hyperparameters in Autoencoder Classification

- **Encoding Dimension:**
    - **Increasing Value:** Retains more features and may capture finer details, but risks including noise and increasing model complexity.
    - **Decreasing Value:** Forces the network to compress information more, which may lose important details if too low, resulting in poor reconstruction quality.
- **Batch Size:**
    - **Increasing Value:** Speeds up training due to efficient computation but can lead to less frequent weight updates and may reduce generalization.
    - **Decreasing Value:** Provides more frequent updates and may help generalization, but slows down training due to higher computational overhead per epoch.
- **Learning Rate:**
    - **Higher Learning Rate:** Can accelerate training but may overshoot the optimal solution, causing instability.

    - **Lower Learning Rate:** Promotes stable convergence but might result in slower training progress.
- **Number of Epochs:**
    - **More Epochs:** Generally improves performance up to a point, but too many can lead to overfitting.
    - **Fewer Epochs:** May result in underfitting if the network hasn't learned enough features from the data.

Things asked in Doc

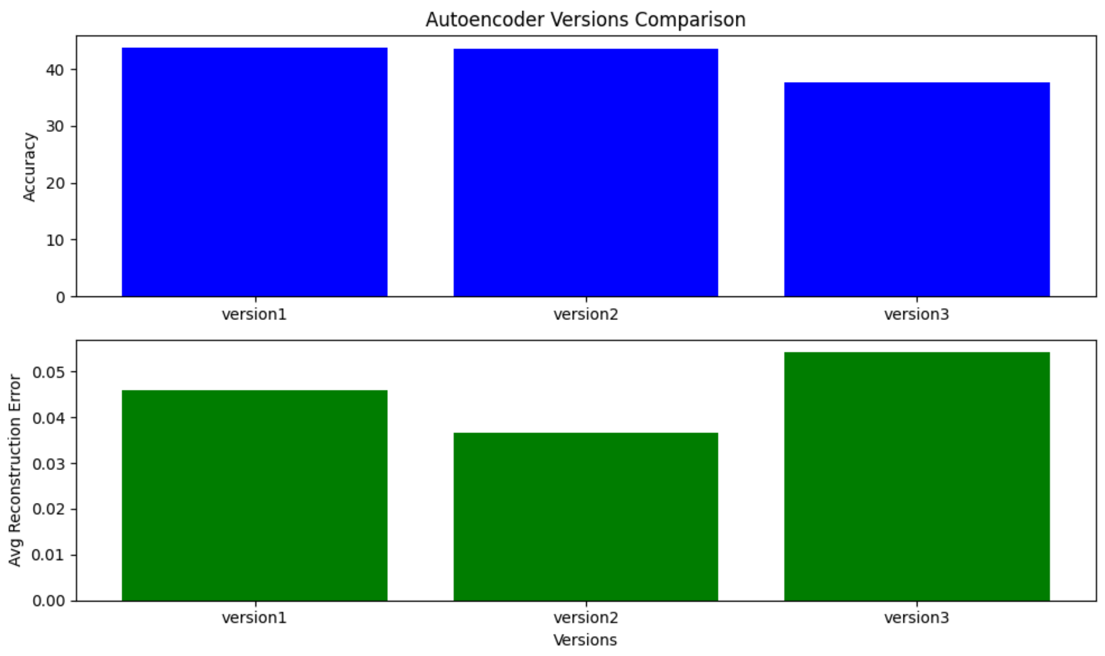
Advantages of Autoencoders:

- **Dimensionality Reduction:**
  Autoencoders compress high-dimensional data into a smaller, more manageable representation, which can speed up subsequent tasks and reduce storage needs.
  Example: Compressing a CIFAR-10 image (3072 features) to a 64-dimensional vector.
- **Denoising:**
  They can remove unwanted noise from data, resulting in cleaner inputs that improve overall performance.
  Example: An autoencoder trained on noisy images produces a clearer, denoised version of a corrupted photograph.
- **Anomaly Detection:**
  Autoencoders learn the typical patterns of normal data, so data that deviates significantly will result in higher reconstruction error, helping to identify anomalies.
  Example: In a production line, defective products produce a high reconstruction error compared to normally produced items.
- **Self-Supervised Feature Learning:**
  They learn useful representations without the need for labeled data, which is valuable when labels are scarce or expensive to obtain.
  Example: An autoencoder extracts features from unlabeled satellite images that can later be used for land cover classification.
- **Smooth Latent Space:**
  Autoencoders create a continuous and smooth latent space, enabling meaningful interpolation between data points.
  Example: Interpolating between two handwritten digits can produce a smooth transition from one digit to another.

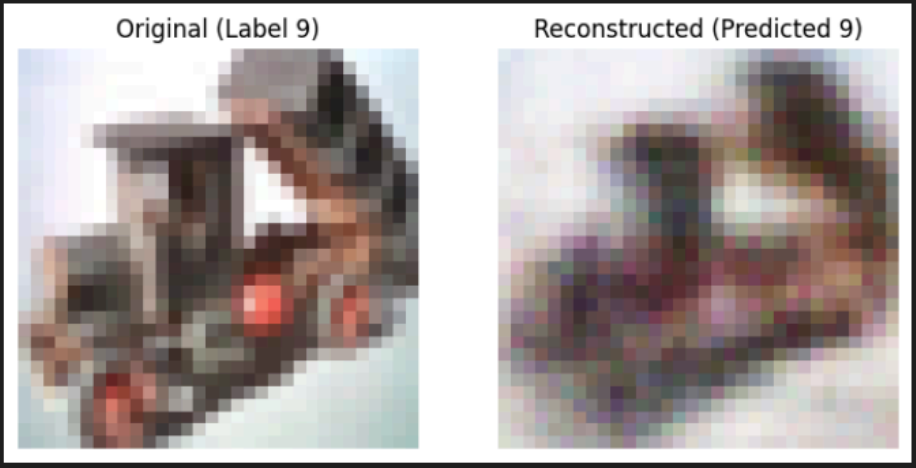
Challenges of Autoencoders:

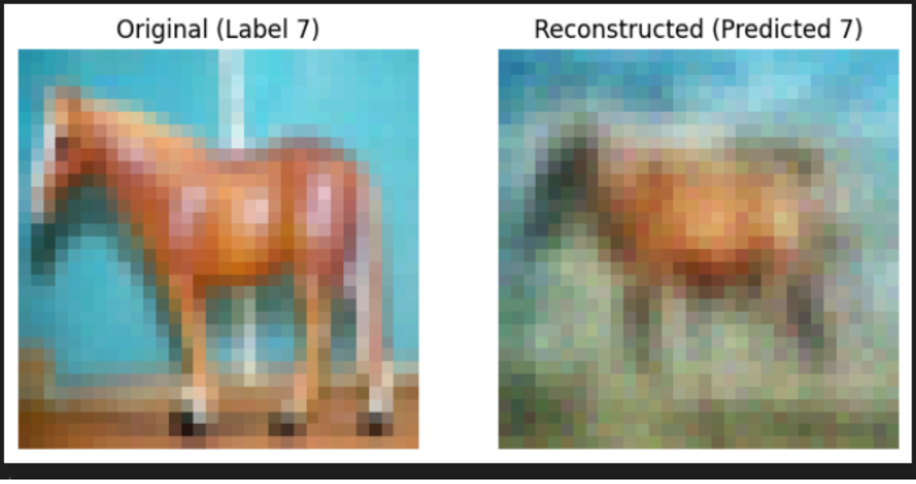
- **Hyperparameter Sensitivity:**
  The performance greatly depends on choices like latent dimension, learning rate, and network depth, requiring careful tuning.
  Example: Setting the latent dimension too low (e.g., 16) may lose important details, while too high (e.g., 256) may lead to overfitting.
- **Overfitting Risk:**
  Complex autoencoder models can memorize the training data instead of learning general features, resulting in poor performance on new data.
  Example: A deep autoencoder might simply copy training images, performing poorly on unseen test images.
- **Lack of Interpretability:**
  The features in the latent space are often abstract and hard to interpret, making it difficult to understand what the model has learned.
  Example: A 64-dimensional latent vector might not directly correspond to clear, identifiable aspects of the original images.
- **Training Instability:**
  Deep or improperly configured autoencoders may suffer from issues like vanishing gradients, leading to unstable training.
  Example: A poorly tuned autoencoder might have inconsistent loss reduction over epochs, hindering convergence.
- **Computational Demand:**
  Training large autoencoders, especially on high-resolution images, can require significant computational resources and memory, posing challenges on limited hardware.
  Example: Running a high-capacity autoencoder on large image datasets may slow down training on a system without a dedicated GPU.

Visualizations



```
Visualizing results for version2
```

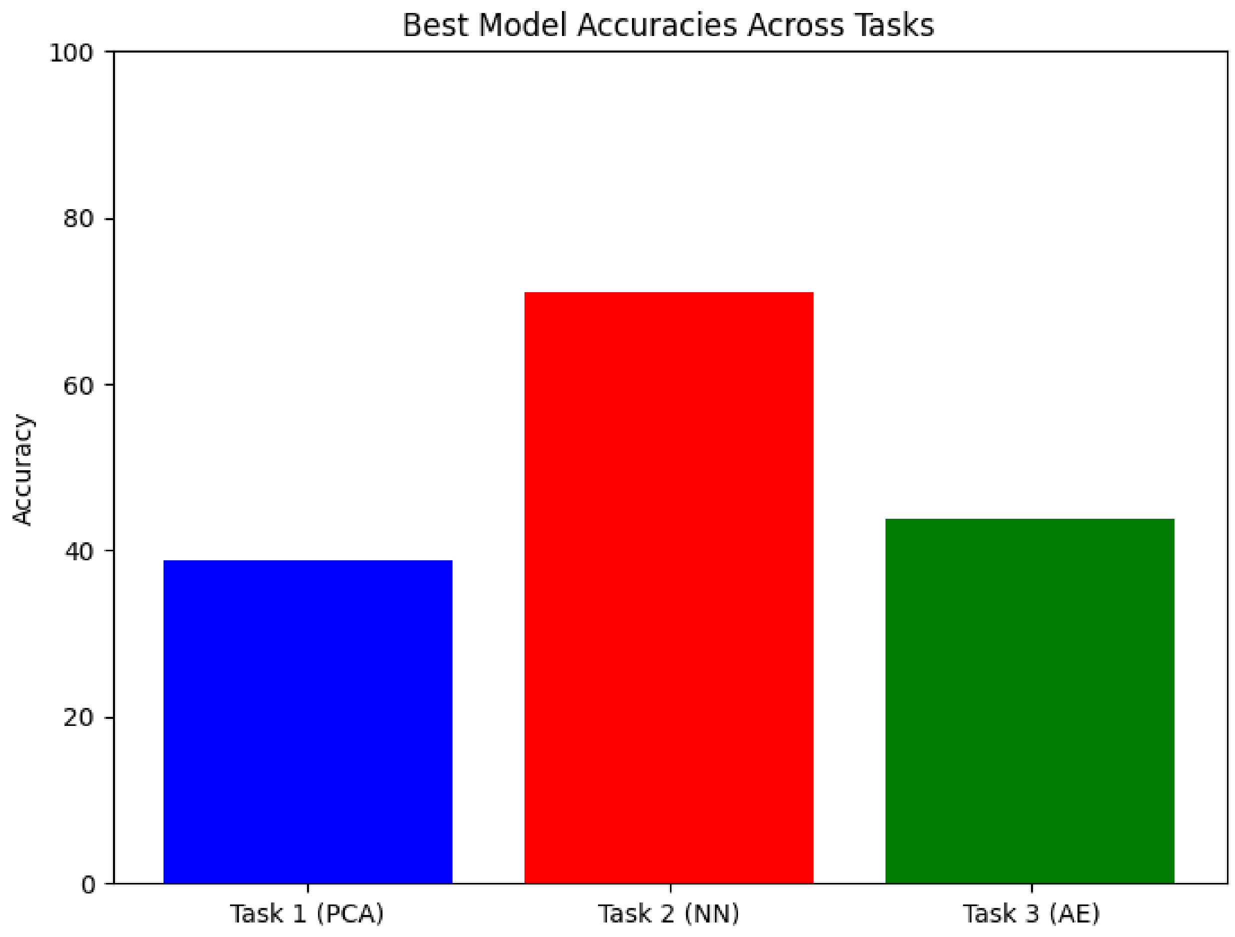


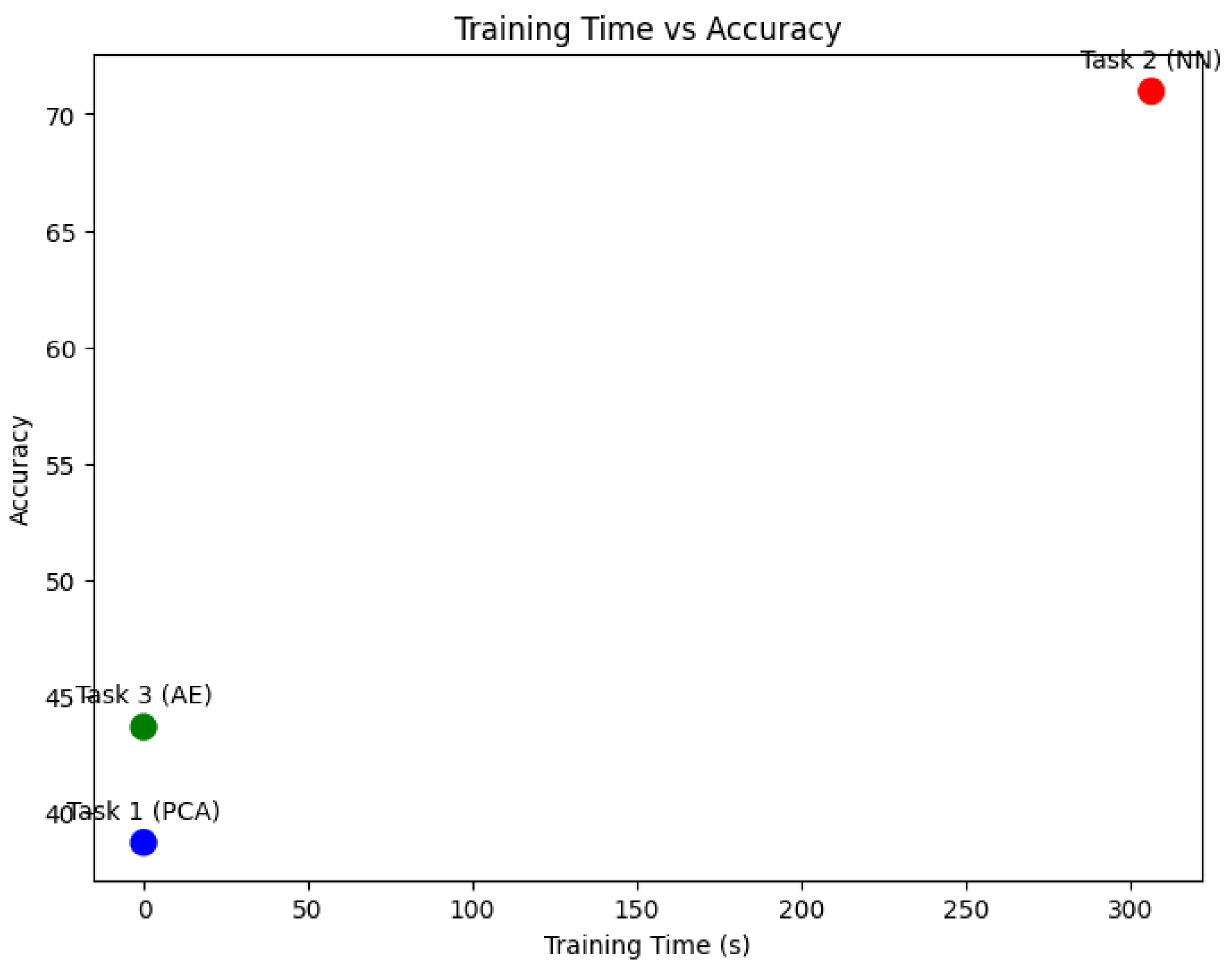


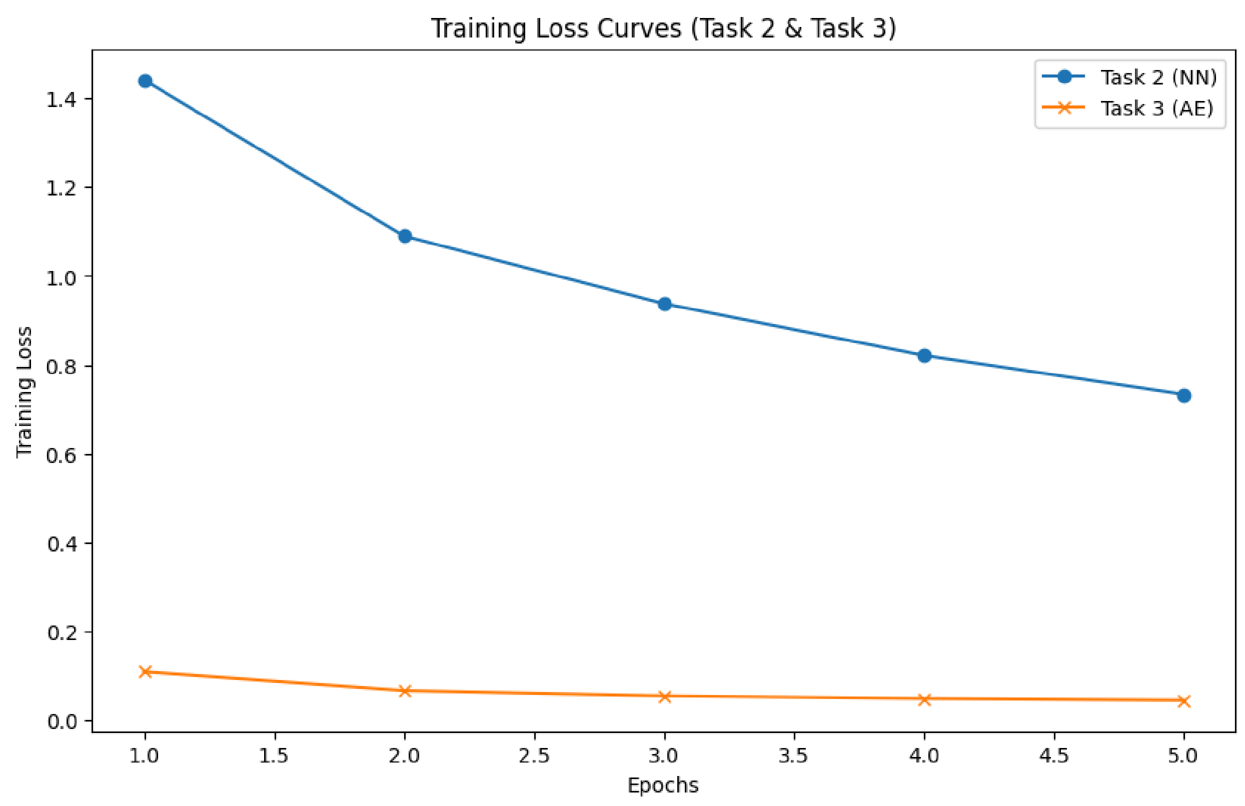
```
The best autoencoder version is version1 with accuracy 43.69
```

Ok so now we will compare all diff methords and find the best one

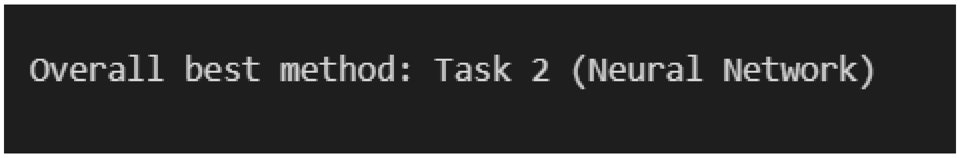
```
Best PCA version (Task 1): version3 Accuracy: 38.73
Best Neural Network version (Task 2): version3 Accuracy: 70.98
Best Autoencoder version (Task 3): version1 Accuracy: 43.69
```




Training Time vs Accuracy
Task 2 (NN)
Task 3 (AE)
Task 1 (PCA)
Accuracy
Training Time (s)
70
65
60
55
50
45
40
0
50
100
150
200
250
300



Neural networks perform best because they capture complex, nonlinear patterns in the data, resulting in higher accuracy even if they take more time to train. In contrast, PCA is too simple it only reduces dimensions, often losing important details while autoencoders can overfit and are harder to tune for optimal performance.



```
Overall best method: Task 2 (Neural Network)
```