This research aims to predict video virality using a multimodal ensemble model. We're leveraging ConvLSTM Autoencoder for video encoding, NLP for audio analysis, and a transformer-based regression model to learn non-linear relationships and predict viewer metrics.

Baseline:

(ResNet) CNN + MLP = MSE Loss: 10:

A combination of a Convolutional Neural Network (CNN) based on the ResNet architecture and a Multi-Layer Perceptron (MLP). The model is trained using Mean Squared Error (MSE) loss, with a loss value of 10.

ResNet + RNN = MSE Loss: 5:

Another variant of the ResNet architecture combined with a Recurrent Neural Network (RNN). The model is trained with MSE loss, yielding a loss value of 5.

I3D (Inflated 3D ConvNet) + LSTM = MSE Loss: 4:

I3D (Inflated 3D ConvNet) is a 3D extension of the ConvNet, along with a Long Short-Term Memory (LSTM) network. The model is trained with MSE loss and achieves a loss value of 4.

Model:

1. The model begins by breaking down the input video into separate visual and audio tracks.

2. **Unsupervised Pretraining - Visual Track**: The visual track of the video is processed using a ConvLSTM Autoencoder. This architecture combines convolutional layers with LSTM (Long Short-Term Memory) layers to capture both spatial and temporal dependencies in the visual data. The autoencoder is trained in an unsupervised manner, meaning it learns to reconstruct the input data without any specific labels. The output of this process is an encoded representation or embedding of the visual context of the video.

**Unsupervised Pretraining - Audio Track**: The audio track of the video is processed using a pretrained model called DeepSpeech, developed by Mozilla or the open-source Whisper. The model converts the audio into a corresponding textual transcript, providing additional information about the content of the video. This step is prebuilt, meaning it uses a pre-existing model rather than training from scratch.

3. **Embedding Generation - Visual Embedding**: The encoded visual representation from the ConvLSTM Autoencoder is extracted as an embedding vector, capturing the visual context of the video.

**Embedding Generation - Audio Embedding**: The DeepSpeech/Whisper model generates textual output from the audio track, representing the spoken content. This output can be considered as an embedding of the audio information.

4. **Concatenation and Feature Fusion**: The visual and audio embeddings are concatenated or combined into a single feature representation and fed into a transformer-based regression model. This fusion of modalities allows the model to leverage both visual and audio information for subsequent analysis. The transformer consists of LSTM or MLP layers.:

5. **Transformer-Based Regression**: The concatenated feature representation is input into a transformer-based regression model. The transformer architecture, commonly used in natural language processing tasks, is adapted here to handle the combined visual and audio data. It learns the semantic and non-linear relationships between the input features and the video creator success metrics, such as video views. The regression model is trained to predict these metrics based on the fused feature representation.

6. **Loss Calculation**: The model is trained using a mean squared error (MSE) loss function, with the goal of minimizing the discrepancy between the predicted values and the actual values. The final MSE loss target is 0.01, indicating a high level of accuracy in predicting the video creator success metrics. The output of the regression model is compared to the ground truth video creator success metrics using the mean squared error (MSE) loss function. The aim is to minimize the difference between the predicted values and the actual values.

7. **Baseline and Evaluation**: To evaluate the effectiveness of the proposed model, a baseline model is established using a simpler architecture or approach. The baseline is compared with the main implementation in terms of the achieved MSE loss. A lower MSE loss value indicates better performance in predicting video creator success metrics.

Overall, this model leverages unsupervised pretraining, pretrained audio models, and a transformer-based regression model to predict video creator success metrics, with a focus on video views. The combination of visual and audio information allows for a more comprehensive understanding of the video content and improves the accuracy of the predictions. The model architecture includes elements of both unsupervised pretraining and supervised fine-tuning, combining the benefits of both approaches. The model also incorporates a process called dual transfer learning, where both the visual and audio embeddings contribute to the final prediction.

Experiment:

Divide the dataset into three parts: training set (70%), validation set (10%), and testing set (20%).

Experiment with different multimodal fusion techniques, such as **early fusion** (combining features before feeding them to the regression model) and **late fusion** (independently training individual models for each modality and then combining their predictions).

**Evaluate the performance** of the model on the testing set using the chosen evaluation metrics (e.g., MSE). Compare the results of different fusion methods to identify the most effective approach.

Conduct **ablation experiments** to study the **impact of each modality** on the prediction performance. Train the model using only video features, only audio features, only textual metadata, and then compare the results to the multimodal fusion approach.

Perform **hyperparameter tuning** to find the **optimal** architecture and parameters for the regression model.

Analyze the model's predictions to gain insights into the **key factors** influencing TikTok content success. **Visualize the attention mechanisms** to understand which parts of the video, audio, or textual information are most relevant for popularity prediction.

Hyperparameter tuning is an essential step in the process of training machine learning models to achieve better performance on a specific task. In the context of the "Multimodal Deep Regression on TikTok Content Success" task, hyperparameter tuning involves finding the optimal values for various hyperparameters that govern the behavior of the deep regression model. The goal is to improve the model's ability to predict the success or popularity of TikTok content accurately.

Here are some key hyperparameters that could be tuned for the multimodal deep regression model:

**Learning Rate**: The learning rate determines the step size at which the model updates its parameters during training. A small learning rate may lead to slow convergence, while a large learning rate may cause the model to overshoot the optimal solution. Commonly used optimization algorithms like Adam and SGD have learning rate parameters that need tuning.

Number of Hidden Layers and Units: The architecture of the deep regression model can greatly impact its performance. This includes determining the number of hidden layers and the number of units (neurons) in each layer. Too few layers or units may result in underfitting, while too many may lead to overfitting.

**Dropout Rate**: Dropout is a regularization technique used to prevent overfitting. It randomly deactivates neurons during training, forcing the model to rely on different pathways for making predictions. The dropout rate determines the probability that a neuron is deactivated.

**Batch Size** (1): The batch size is the number of samples used in each iteration of the training process. A larger batch size can lead to faster convergence but may require more memory, while a smaller batch size can increase training time but may lead to better generalization.

**Number of Epochs**: The number of epochs defines how many times the model will iterate over the entire dataset during training. Too few epochs may result in underfitting, while too many may lead to overfitting.

**Regularization Strength** (Transformer regularization L1 and L2

): L1 and L2 regularization can be applied to the model to penalize large weights. The regularization strength determines the impact of the regularization term on the loss function.

Activation Functions: The choice of activation functions in the model's hidden layers can impact its ability to capture complex patterns in the data. Common choices include ReLU, tanh, and sigmoid functions.

Image and Video Preprocessing: The preprocessing steps for images and videos may include resizing, cropping, normalization, and data augmentation. These steps can also have an impact on the model's performance.

**Define the Search Space**: Before starting the hyperparameter tuning, it is essential to define the search space for each hyperparameter. For example, the learning rate could be searched in a range from 0.0001 to 0.1, the batch size could be chosen from [32, 64, 128], and the number of layers could be in the range [1, 3, 5]. Each hyperparameter should have a predefined set of possible values or distributions.

To tune these hyperparameters, there are several techniques.

* **Grid Search**: It exhaustively searches all possible combinations of hyperparameters from the predefined search space. While it's straightforward, it could be computationally expensive and time-consuming.
* **Random Search**: Randomly samples hyperparameters from the predefined search space. It is less computationally intensive compared to grid search and often yields similar results.
* **Bayesian Optimization**: Bayesian optimization builds a probabilistic model of the objective function and chooses the next hyperparameter combination based on the previous observations. It is often more efficient in finding optimal hyperparameters compared to grid or random search.
* **Genetic Algorithms**: Genetic algorithms use techniques inspired by natural selection to evolve a population of hyperparameter combinations over generations.

**Early Stopping**: Early stopping is a technique used during training to prevent overfitting. It stops training the model when the performance on the validation set starts to degrade.

During hyperparameter tuning, it is crucial to use proper validation techniques like cross-validation to avoid overfitting to the validation set. The process may involve multiple iterations of training and validation until the best combination of hyperparameters is found. Once the optimal hyperparameters are determined, the model can be trained on the entire dataset using these values to make predictions on new TikTok content and estimate their success.

**Regularized regression** is a variant of linear regression that introduces limitations or corrections to the coefficient estimates to avoid them from approaching zero. Essentially, regularization helps reduce the computational burden during model operation and mitigates non-uniform stability issues that may arise during the learning process. By doing so, it effectively eliminates the risk of overfitting. The regularization function incorporates additional terms into the original loss function, effectively acting as a penalty for the complexity of the model.

In this context, L2 regularization is specifically employed to combat overfitting and enhance the generalization capability of the model. It ensures that the model's parameters do not become overfitted and helps reduce the risk of overfitting. By adjusting the parameter 𝛼 in the L2 regularization term, the weight matrix of the model is controlled. A higher 𝛼 value results in smaller weights for many hidden elements in the model, making their impact negligible. This process essentially reduces the neural network to a smaller version while maintaining the network's depth. As a result, the model transitions from an overfitting state to an underfitting state.

In the context of human action recognition, the L2 regular joint attention mechanism is employed. L2 weight decay regularization is a common technique used in deep learning, particularly to address overfitting issues caused by an excessive number of features. The solution involves reducing the weight of features or penalizing unimportant features. Regularization can effectively prevent overfitting and enhance the model's ability to generalize.

When training the TimeSformer model, applying L2 weight decay regularization can prevent overfitting on the training data and improve the model's generalization capabilities. Specifically, L2 regularization introduces the L2 norm into the loss function, which penalizes parameters and achieves the desired regularization effect.

A diagram and a diagram of a function

Description automatically generated

\*\*Most Important\*\* What is difference in performance between **early** vs **late fusion**, which is better, what study suggest that? (It is late fusion for this project)

Early fusion and late fusion are two different approaches to combining information from multiple sources or modalities in a machine learning model. The difference between early and late fusion and the performance comparison are presented below.

**Early fusion** involves combining the features or representations from different modalities at an early stage of the model. This means that the data from different sources is concatenated or merged before feeding it into the main model, and the subsequent layers process the fused data. Early fusion can benefit from joint feature learning across modalities but may face challenges when modalities have different levels of significance or when certain modalities are more noisy than others. The general scheme for early fusion is illustrated in the following figure.

A diagram of a software process

Description automatically generated

**Figure1** General scheme for early fusion. Output of unimodal analysis is fused before a concept is learned. *(Patel, K., Chellappa, R., & Phillips, P. J. (2006). Early versus late fusion in semantic video analysis. In Proceedings of the 14th ACM international conference on Multimedia (pp. 869-878))*

**Late fusion** involves processing the data from different modalities separately through individual models and then combining their predictions or representations at a later stage. This allows each modality to be processed independently, possibly capturing unique patterns and reducing the risk of interference between modalities. However, late fusion might not fully leverage the joint information from all modalities. The general scheme for late fusion is illustrated in the following figure.

A diagram of a computer process

Description automatically generated

**Figure 2** General scheme for late fusion. Output of unimodal analysis is used to learn separate scores for a concept. After fusion a final score is learned for the concept. *(Patel, K., Chellappa, R., & Phillips, P. J. (2006). Early versus late fusion in semantic video analysis. In Proceedings of the 14th ACM international conference on Multimedia (pp. 869-878))*

The performance of early fusion versus late fusion can depend on various factors, e.g., the nature of the multimodal data, the complexity of the task, and the level of interdependence between modalities. It is important to note that the performance comparison between early and late fusion methods can also be influenced by other factors such as the quality of the features, the architecture of the network, the size of the dataset, and the availability of labeled data. It is usually necessary to experiment with both approaches and analyze their results to determine the optimal fusion strategy for a given task. Therefore, there is no definitive answer as to which fusion approach is always better. In some cases, early fusion may be more effective, while in others, late fusion may yield superior results.

Based an experiment on 184 hours of broadcast video using 20 semantic concepts, the detection results for both early fusion and late fusion are visualized in the following figure *(Patel, K., Chellappa, R., & Phillips, P. J. (2006). Early versus late fusion in semantic video analysis. In Proceedings of the 14th ACM international conference on Multimedia (pp. 869-878))*. From the figure below, it is evident that late fusion outperforms early fusion in numerous concepts such as golf, boat, and ice hockey. However, late fusion's performance is comparatively weaker in certain concepts, like car (with an absolute difference of 0.1) and stock quotes (with an absolute difference of 0.3).

A graph with red and blue dots

Description automatically generated

**Figure 3** Comparison of early fusion versus late fusion for semantic indexing of 20 concepts. *(Patel, K., Chellappa, R., & Phillips, P. J. (2006). Early versus late fusion in semantic video analysis. In Proceedings of the 14th ACM international conference on Multimedia (pp. 869-878))*

# *Patel, K., Chellappa, R., & Phillips, P. J. (2006). Early versus late fusion in semantic video analysis. In Proceedings of the 14th ACM international conference on Multimedia (pp. 869-878).*

# [*https://dl.acm.org/doi/10.1145/1101149.1101236*](https://dl.acm.org/doi/10.1145/1101149.1101236)

# This paper compares early and late fusion techniques in the context of semantic video analysis. Based on an experiment on 184 hours of broadcast video using 20 semantic concepts, it concludes that late fusion methods are often superior for tasks that involve diverse and complex multimodal information.

# *Zhou, T., Zhang, Z., Chen, W., & Chen, Z. (2021). Early, intermediate and late fusion strategies for robust deep learning-based multimodal action recognition. Signal, Image and Video Processing, 15(7), 1709-1716.*

# [*https://link.springer.com/article/10.1007/s00138-021-01249-8*](https://link.springer.com/article/10.1007/s00138-021-01249-8)

# This study explores early, intermediate, and late fusion strategies for multimodal action recognition. It shows that late fusion methods tend to achieve better performance in certain cases, but the choice of fusion strategy varies based on the specific multimodal features and the complexity of the action recognition task.

# *Xia, S., Xu, H., & Wang, C. (2022). Early or Late Fusion Matters: Efficient RGB-D Fusion in Vision Transformers for 3D Object Recognition. arXiv preprint arXiv:2210.00843.*

# <https://arxiv.org/pdf/2210.00843.pdf>

# This work investigates early and late fusion in vision transformers for 3D object recognition with RGB-D data. It demonstrates that early fusion leads to better performance when combining RGB and depth information in this specific task.

# *Malik, A., & Wu, F. (2020). Early vs Late Fusion in Multimodal Convolutional Neural Networks. In 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (pp. 3547-3551). IEEE.*

# [*https://ieeexplore.ieee.org/document/9190246*](https://ieeexplore.ieee.org/document/9190246)

# The paper explores early and late fusion in multimodal convolutional neural networks and concludes that late fusion methods generally perform better for tasks involving multiple modalities

*Zadeh, A., Chen, M., Poria, S., Cambria, E., & Morency, L. P. (2018). Multi-attention recurrent network for human communication comprehension. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2018.*

[*https://arxiv.org/pdf/1802.00923.pdf*](https://arxiv.org/pdf/1802.00923.pdf)

This paper compares early fusion and late fusion approaches for multimodal sentiment analysis. They propose a multi-attention recurrent network that can effectively integrate text and visual modalities. The results of their experiments suggest that the performance of early fusion and late fusion can vary depending on the task and dataset.

How is **batch size** of 1 affect optimization?

Using a batch size of 1 in the optimization of a neural network can have both advantages and disadvantages. The trade-offs between large and small batch sizes, and the challenges associated with very small batch sizes are presented below. The choice of batch size depends on the specific problem, available hardware, and the trade-offs one is willing to accept. Smaller batch sizes may be preferred for certain applications where generalization and exploration are crucial, while larger batch sizes may be more suitable when computational efficiency and stable convergence are prioritized. It is essential to experiment with different batch sizes and monitor the training progress to determine the optimal value for a particular task. Hyperparameter tuning techniques and learning rate schedules can also be used to optimize the training process further.

***Advantages:***

**Faster Updates**: With a batch size of 1, each update to the model's parameters is based on a single data point. This means that the updates can be faster since there are fewer computations involved compared to larger batch sizes. This advantage is especially prominent when training on large datasets.

**More Frequent Weight Updates**: Smaller batch sizes allow for more frequent weight updates. This can be beneficial as the model can adapt quickly to changes in the data distribution.

**Exploration**: Smaller batch sizes can promote more exploration in the weight space, allowing the model to escape from local minima and possibly find better optima.

***Disadvantages:***

**Noisy Updates**: The updates based on single data points are more noisy and have higher variance. This can result in unstable training and hinder convergence.

**Slower Convergence**: Due to the noisy updates, the training process might take more time to converge to an optimal solution. This is because the model might oscillate around the optimal solution or experience difficulties in finding it.

**Overfitting**: With a batch size of 1, the model might memorize the training data rather than learning general patterns, leading to overfitting.

**Instability**: A batch size of 1 can lead to unstable gradients, especially in deeper networks, potentially causing the model to diverge during training.

*Masters, D., & Luschi, C. (2018). Revisiting Small Batch Training for Deep Neural Networks.*

# <https://arxiv.org/pdf/1804.07612.pdf>

*Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., ... & Jouppi, N. P. (2017). Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour.*

# <https://arxiv.org/pdf/1706.02677.pdf>

How to **optimize swin transformer** specifically, be very specific?

Here are some common methods for optimizing transformers, including the Swin Transformer.

**Adam Optimizer**: Adam (Adaptive Moment Estimation) is a widely used optimization algorithm for training deep learning models, including transformers like Swin Transformer. It combines the benefits of both AdaGrad and RMSprop and adapts the learning rate for each parameter during training.

**Learning Rate Scheduling**: Learning rate scheduling is often employed to adjust the learning rate during training. Techniques like warm-up, step decay, or cosine annealing can help stabilize training and achieve better performance.

**Gradient Clipping**: Gradient clipping is a technique used to prevent gradients from becoming too large during training, which can help prevent exploding gradients and improve training stability.

**Weight Decay**: Regularization techniques like weight decay can be used to penalize large weights in the model, which can improve generalization and prevent overfitting.

**Data Augmentation**: Data augmentation is essential for enhancing the diversity of the training data and can help the model generalize better.

The code provided in the following Github link is to build and set up the optimizer for training the Swin Transformer. It supports different optimization algorithms such as SGD, AdamW, FusedAdam, and FusedLAMB. The optimizer is also configured to handle weight decay for different parameter groups in the model, allowing fine-tuning and pre-training scenarios. The summary of how the Swin Transformer was optimized based on the code is presented below.

**Weight Decay Handling:** The model's parameters are divided into two groups: one with weight decay and the other without weight decay. The weight decay is set to 0 for the parameters specified in the **no\_decay** group.

**Parameter Grouping:** The parameters are grouped based on their names and properties. Parameters that do not require weight decay (e.g., bias terms) or specified in the **skip\_list** or **skip\_keywords** are added to the **no\_decay** group.

**Learning Rate Scheduling:** The learning rate can be scaled differently for different layers of the Swin Transformer. The **get\_swin\_layer** function determines the layer id based on the parameter name, and the learning rate is scaled using the **scales** parameter if provided.

**Supported Optimizers:** The code supports several optimizers, including SGD, AdamW, FusedAdam, and FusedLAMB, and the chosen optimizer is initialized accordingly.

References (the original Swin Transformer paper, which introduced the model architecture and provided details on how it was trained and optimized):

*Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., ... & Huang, T. S. (2021). Swin Transformer: Hierarchical Vision Transformer using Shifted Windows. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).*

[*https://arxiv.org/pdf/2103.14030.pdf*](https://arxiv.org/pdf/2103.14030.pdf)

<https://github.com/microsoft/Swin-Transformer>

About the **optimization techniques**, which order shall we apply specifically? How we narrow down the effective range?

When applying optimization techniques to train a deep learning model, including the Swin Transformer, the order of operations and the effective range of hyperparameters can significantly impact training performance. Here's a suggested order of applying optimization techniques, along with information for narrowing down the effective hyperparameter range:

1. **Data Augmentation:** Data augmentation is essential to enhance the diversity of the training data and improve the model's generalization. Common augmentation techniques for image data include random cropping, flipping, rotation, color jittering, and random resizing.
2. **Learning Rate Scheduling:** It is often recommended to start training with a relatively high learning rate and gradually decrease it during the training process. Common approaches include warm-up, step decay, or cosine annealing. Warm-up involves using a lower learning rate initially and then gradually increasing it. Step decay involves reducing the learning rate at predefined epochs, while cosine annealing smoothly reduces the learning rate over time. The specific schedule and parameters can be tuned using validation data.
3. **Weight Decay:** Weight decay is a regularization technique to prevent overfitting. It is typically applied to the weights of the model's parameters during optimization. It is crucial to select an appropriate weight decay value, which can be achieved through experimentation. Common values include 1e-4 or 1e-5.
4. **Gradient Clipping:** Gradient clipping is applied to prevent gradients from becoming too large during optimization, which can lead to instability. It helps maintain stable updates and prevents exploding gradients. A common clipping threshold is 1.0 or 5.0.
5. **Batch Size and Batch Normalization:** Batch size is a crucial hyperparameter that can affect optimization. Larger batch sizes can lead to more stable updates, but they may require more memory and can slow down convergence. Smaller batch sizes can improve generalization but may introduce more noise and slow down training. Batch Normalization can also be beneficial for normalization but may not be as crucial for Swin Transformers due to their window-based positional encoding. By using Batch Normalization, we can set the learning rates high which speeds up the training process.
6. **Optimizer Choice**: Common choices include AdamW or SGD with momentum. The learning rate and other hyperparameters of the optimizer can be fine-tuned using validation data.
7. **Data Balancing**: If the dataset is imbalanced, consider applying data balancing techniques, such as weighted loss functions or oversampling/undersampling, to handle class imbalances.
8. **Early Stopping**: Implement early stopping to prevent overfitting and save computational resources. Monitor the validation loss and stop training when it starts to increase or stagnate.

To narrow down the effective range for each optimization technique, it is important to perform hyperparameter tuning experiments using a validation dataset. For each technique (e.g., learning rate, weight decay, batch size), try different values within a predefined range and track the model's performance on the validation set. Based on the results, select the optimal hyperparameters that lead to the best performance for your specific task. Techniques such as grid search, random search, or Bayesian optimization can be used to efficiently explore the hyperparameter space and find the optimal combination of hyperparameters.

*Smith, S. L., & Le, Q. V. (2018). A Disciplined Approach to Neural Network Hyper-Parameters: Part 1 – Learning Rate, Batch Size, Momentum, and Weight Decay. arXiv preprint arXiv:1803.09820.*

# <https://arxiv.org/pdf/1803.09820.pdf>

*Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., ... & Jouppi, N. P. (2017). Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. arXiv preprint arXiv:1706.02677.*

# <https://arxiv.org/pdf/1706.02677.pdf>

**The\_Multimodal\_Final.py** isan implementation of the multimodal deep regression model using PyTorch. The model includes building an autoencoder (ConvLSTMAutoencoder) to process visual data and an ensemble model that combines visual and audio data using Transformer-based models. Here's a breakdown of the code:

1. **Virtual Environment Setup**: This section provides instructions on how to set up a virtual environment for the project and install the required dependencies. It recommends creating a virtual environment named "multimodal" outside the project folder to keep dependencies isolated.
2. **Importing Libraries**: This section imports the necessary Python libraries for building and training the deep regression model. The key libraries used are:
   * **torch**: The main PyTorch library for building and training neural networks.
   * **torch.nn**: Provides various neural network modules and loss functions.
   * **torch.optim**: Contains different optimization algorithms, such as Adam used in this code.
   * **DataLoader**: Used for loading data in batches during training.
   * **tqdm**: A library for displaying progress bars during long-running computations.
   * **scikit-learn**: Used for data preprocessing and evaluation metrics like Mean Squared Error (MSE).
   * **matplotlib.pyplot**: Used for plotting graphs and visualization.
   * Other standard Python libraries like **os**, **random**, **time**, **whisper**, **numpy**, and **warnings**.
3. **Data Preprocessing**: This section involves several steps to preprocess the data for training the multimodal deep regression model. It includes the following steps:
   * Extracting audio from video datasets and saving them to the audio directory.
   * Using Whisper library to transcribe audio dialog and extract Long-Short Term Memory (LSTM) embeddings.
   * Processing visual data, including parameters like frame skip, shrink scale, and normalization. The processed visual tensors are saved to directories and later loaded for training and validation.
4. **ConvLSTMAutoencoder**: The code defines a ConvLSTMAutoencoder model. An autoencoder is a type of neural network that is used for unsupervised learning to learn efficient representations of the input data. In this case, the ConvLSTMAutoencoder is designed to process and learn efficient representations of visual data.
5. **Training the ConvLSTMAutoencoder**: This section involves training the ConvLSTMAutoencoder model using Mean Squared Error (MSE) loss and the Adam optimizer. The training is performed for a specified number of epochs, and training and validation losses are recorded for later visualization.
6. **Ensemble Model**: The code defines the EnsembleModel, which is used to combine the visual and audio data for the multimodal deep regression. The ensemble model likely combines the outputs of the TransformerModel\_Visual and TransformerModel\_Audio models.
7. **Training the Ensemble Model**: This section involves training the EnsembleModel. The visual data is passed through the trained ConvLSTMAutoencoder to obtain embeddings before being combined with the audio embeddings from the TransformerModel\_Audio. The ensemble model is trained on the combined data using MSE loss and the Adam optimizer.
8. **Validation and Evaluation**: The code evaluates the trained Ensemble Model on the validation set. It calculates the Mean Squared Error (MSE) between the predicted values and the ground truth values, providing an indication of the model's performance.
9. **Plotting**: This section generates plots for the training and validation loss during the training of the ConvLSTMAutoencoder and Ensemble Model. The loss plots help visualize how the model's performance improves over the training epochs.
10. **Inspecting Results**: The code inspects and compares the ground truth and predicted values for the validation set. It plots the actual and predicted values for a random sample from the validation set and provides the Mean Squared Error (MSE) value between the two sets of values.

**Transformer.py** defines two components of a multimodal deep regression model using Transformers: **TransformerModel\_Visual** and **TransformerModel\_Audio**. The model architecture is inspired by the Transformer model introduced in the paper "Attention Is All You Need" (https://arxiv.org/abs/1706.03762). The model also includes a **PositionalEncoding** module for adding positional encodings to the input data.

1. **PositionalEncoding**: This module is used to add positional encodings to the input data to provide positional information to the Transformer model. The positional encodings are calculated using sine and cosine functions with varying frequencies.
2. **TransformerModel\_Visual**: This class defines the visual part of the multimodal deep regression model. It takes as input the video embeddings and processes them through a Transformer Encoder. It first applies positional encodings using the **PositionalEncoding** module and then passes the encoded data through a TransformerEncoder composed of multiple **TransformerEncoderLayer** layers. After the Transformer encoding, the output is flattened and passed through a linear layer to obtain the final visual embeddings.
3. **GaussianNormalization**: This module is used to normalize the audio embeddings using Gaussian normalization. It shifts the embeddings to have a mean of 0 and a standard deviation of 1.
4. **TransformerModel\_Audio**: This class defines the audio part of the multimodal deep regression model. It takes as input the audio embeddings and processes them through either a Transformer Encoder (if **pass\_transformer** is set to False) or directly through the Gaussian normalization (if **pass\_transformer** is set to True). If the **pass\_transformer** flag is set to False, the audio embeddings are first passed through the same TransformerEncoder structure as used in **TransformerModel\_Visual**, including positional encodings and multiple Transformer layers. After the encoding, the output is flattened and passed through a linear layer to obtain the final audio embeddings. If **pass\_transformer** is set to True, the audio embeddings are directly passed through the Gaussian normalization.

The implement is to create two different Transformer models, one for processing visual data and the other for processing audio data. The output embeddings from both models can then be combined or used separately for multimodal regression tasks, where the model predicts target value based on both visual and audio data.

**Swin\_Transformer** is a PyTorch model called Swin\_Transformer\_model, which uses the Swin Transformer architecture for processing 3D data. The SwinTransformer3D is an extension of the original Swin Transformer for image processing tasks, adapted for 3D data like videos or volumetric data. The Swin Transformer is a state-of-the-art architecture known for its efficient and powerful attention mechanisms, making it suitable for processing 3D data efficiently. This model takes advantage of the Swin Transformer's capabilities and performs regression on the input data, aiming to predict a single scalar value as the output.