ASSIGNMENT 2

Facial Emotion Recognition with Data Augmentation

Advanced Machine Learning (AL_KSAIM_9_1)

MSc in Software Design with Artificial Intelligence
Technological University of the Shannon

A00315339: Vasyl Dykun
wasyldykun@gmail.com

2024

Contents

Introduction

As part of my Advanced Machine Learning course, I have been tasked with developing a Convolutional Neural Network (CNN) capable of recognizing facial emotions. For this task, I will be utilizing the "Facial Expression Recognition (FER2013) dataset", as specified by the assignment requirements. This paper serves as a report where I discuss the design pathway, experimental results, and findings of my project.

The complete pipeline, from data exploration to model evaluation, is documented in the attached Jupyter notebook.

Data Exploration

The 'Facial Expression Recognition (FER2013)' dataset is available through the following link: https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/data. According to its description, it comprises images of faces, each measuring 48x48 pixels [1]. Every face is categorized into one of seven facial expressions [1]. The categories are as follows:

- 0=Angry
- 1=Disgust
- 2=Fear
- 3=Happy
- 4=Sad
- 5=Surprise
- 6=Neutral

For the project, 'train.csv' dataset was uploaded using Pandas. The dataset dimensions are [28709, 2]. Several samples illustrated in Figure 1 and their visuals are in Figure 2:

	emotion	pixels
18082	3	184 183 193 146 46 44 45 47 45 42 49 44 51 41 ...
20293	5	18 20 28 42 86 97 68 92 132 151 155 162 157 16...
19971	3	255 179 92 92 100 120 138 145 161 176 182 189 ...
13023	5	254 255 215 63 58 58 51 36 20 26 59 126 178 18...
23361	0	255 254 254 253 255 253 239 236 248 238 218 22...

Figure 1. Samples from the dataset

Figure 2. Samples visual representations

Figure 3 presents the distribution of the 'emotion' category within the dataset. Analysis of the distribution indicates the following:

- The distribution across categories is uneven, featuring two outliers: '3 - Happy,' which contains 7215 instances, and '1 - Disgust,' with only 436 instances.
- The distribution among the other categories is more balanced: '6 - Neutral' includes 4965 instances, closely followed by '4 - Sad' with 4830 instances; '2 - Fear' has 4097 instances, closely matching '0 - Angry,' which has 3995 instances; '5 - Surprise' is somewhat isolated with 3171 instances.
- **The dataset exhibits class imbalance, necessitating strategies such as class weighting or others to mitigate potential model bias.**

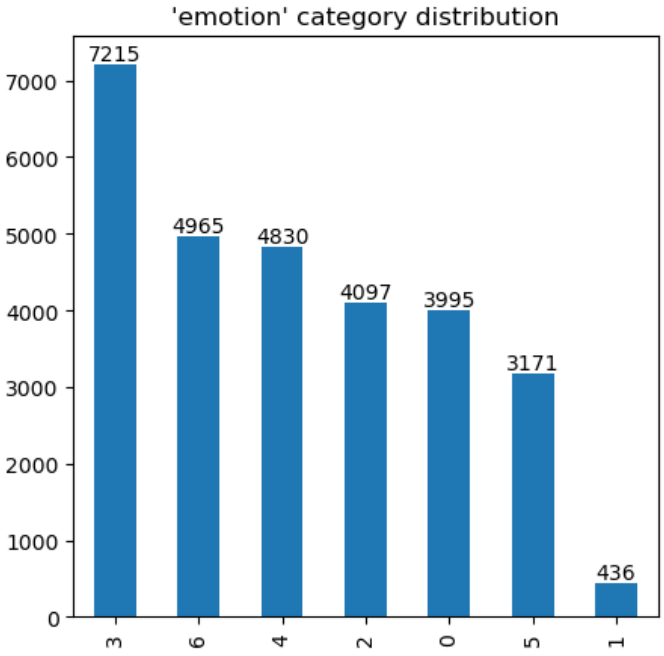


Figure 3. Distribution of 'emotion' category in the dataset

Data preprocessing

The initial step in data preprocessing involved converting a Pandas dataset into a PyTorch Dataset object. To accomplish this, I created a custom class that inherits from torch.utils.data.Dataset. This class follows the standard initialization and implementation pattern but adds functionality to transform the string text in the 'pixels' column into a numeric matrix of size 48x48x1. The code of the class is illustrated in Figure 4.

To normalize the numeric values to the range [0, 1], the class utilizes transformers. The complete implementation of these transformers is detailed in the following section of the report.

```python
# Prepare custom PyTorch Dataset class for train and test sets
class FER_Dataset(Dataset):
  def __init__(self, data, transform=None):
      # for image convert str values into 1x48x48 matrixes
      self.images = data.iloc[:, 1].apply(lambda x: np.array(x.split(' '), dtype=np.uint8).reshape(-1, 48, 48))
      self.images = tuple([torch.tensor(x) for x in self.images])
      self.labels = data.iloc[:, 0]
      self.labels = tuple([torch.tensor(x) for x in self.labels])
      self.transform = transform

  def __len__(self):
      return len(self.images)

  def __getitem__(self, idx):
      image = self.images[idx]
      label = self.labels[idx]

      if self.transform:
          image = self.transform(image)

      return image, label
```

Figure 4. Custom PyTorch Dataset class

For dividing the dataset into training and testing sets, I employed the 'sklearn.model_selection.train_test_split function'. This function was executed with the stratify parameter set to the dataset's 'emotion' category, ensuring that both the test and train datasets exhibit identical class distribution.

Given the dataset's class imbalance and to counter potential complications it might introduce, I implemented sample weights for the training set. These are utilized in conjunction with torch.utils.data.WeightedRandomSampler." (Figure 5)

```python
# Generate sample weights for training set
# get amount of samples in each category
class_counts = dict(train_set.emotion.value_counts())
# create sample weights
sample_weights = [1.0 / class_counts[w] for w in train_set.emotion]
# Create weighted sampler
weighted_sampler = WeightedRandomSampler(sample_weights, len(sample_weights), replacement=True)
```

Figure 5. Weighted Random Sampler

Data Augmentation

To determine the most effective data augmentation strategy, I referred to the paper 'Image Data Augmentation Approaches: Comprehensive Survey and Future Directions' [2] as well as the PyTorch documentation.

For this project, I tested the following augmentation techniques, which include:

- rotation
- translation
- scaling
- shearing
- horizontal flipping
- random erasing
- colour jittering

As suggested in the PyTorch documentation, I utilized PyTorch Transformers v2.

During the experimental phase, I encountered several challenges: an overly large number of transformers significantly slows down the training process, while excessively high

transformation values prevent the model from training. Conversely, if the values are too small, the model begins overfitting. After manually adjusting the settings, I compiled a list of transformers and their parameters that yielded promising results, as shown in Figure 6.

```
# Transformer for training set with data augmentation
transformer_train = v2.Compose([
  v2.RandomHorizontalFlip(p=0.5),          # 50% to flip horisontally
  v2.RandomAffine(degrees=8,               # Randomly rotated up to 8 angle
                  translate=(0.2, 0.2),    # Randomly shifts by x or y up to 20%
                  scale=(0.8, 1.2)),       # Randomly scales in range 20%
  v2.ToDtype(torch.float32, scale=True),   # Transfrom into float32 tensor scaled to [0, 1]
  v2.Normalize((0.5077,), (0.2550,)),      # Normalise image
])

# Transformer for test set
transformer_test = v2.Compose([
  v2.ToDtype(torch.float32, scale=True),   # Transfrom into float32 tensor scaled to [0, 1]
  v2.Normalize((0.5077,), (0.2550,)),      # Normalise image
])
```

Figure 6. Training set transformers code

Model Development

Model structure

During the model development process, I experimented with various CNN layouts and parameters:

- The experiments demonstrated that while simple 2–4-layer CNNs operate at high speeds, they fall short in accuracy. The highest accuracy attained with these networks did not surpass 45%.
- Attempts to employ wide CNN networks with 64 outputs in the first layer revealed that the resulting models operate extremely slowly and are unsuitable for the available computational resources.
- Deep networks with 16 outputs in the first layer, but consisting of 7 or more layers, showed promising accuracy and were significantly faster than their wide counterparts. Therefore, they were selected for this project.

The deep networks encountered issues with the potential for vanishing gradient problems. To address this, I utilised the Residual Networks architecture, which are effective solution for this type of problems [4]. The structure of the created model mirrors the typical layout of conventional Residual Networks, beginning with a single convolution layer, followed by several sets of residual layers, and concluding with two fully connected linear layers. Each convolution layer includes batch normalization, and the last convolution layer features dropout. These strategies improve the training process and help prevent overfitting. The model's weights were initialized using the Kaiming normal distribution.

Experiments with activation functions revealed that LeakyReLU outperforms ReLU:

- The model with LeakyReLU achieved a 2% improvement in performance in relation to its counterpart.
- The model with ReLU exhibited an unusual anomaly where it failed to recognize any instances of the class ‘2=fear’. I suspect this may be related to the "dying ReLU"

phenomenon, in which neurons utilizing ReLU as the activation function become inactive during training.

The model's pipeline is shown in Figure 7, and the full model layout in text format is shown in Appendix 1.

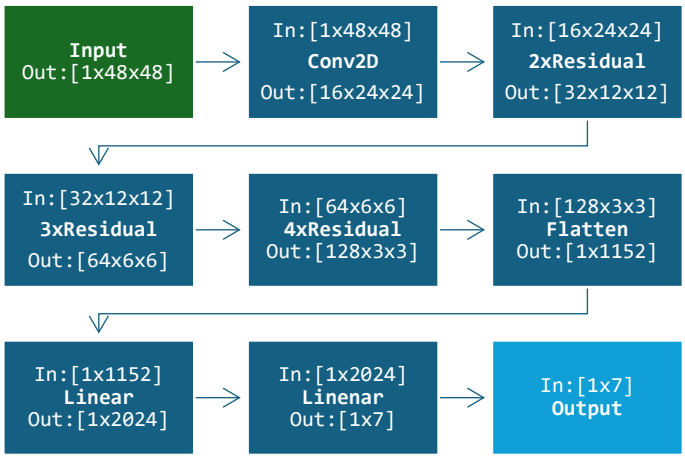


Figure 7. The models pipeline

Model loss, optimisation, and evaluation

The model employs Cross Entropy Loss for calculating the loss function. For optimization, it utilizes the Adam algorithm with a learning rate of '0.0001'. Experiments indicate that a higher learning rate destabilizes the training process. To dynamically adjust the learning rate, the model incorporates the PyTorch ReduceLROnPlateau learning rate scheduler.

For evaluation metrics, the model utilizes three parameters:

- **Training Loss:** This is the Cross Entropy Loss measured between the predicted values of the training set and the actual values. It is utilized for backpropagation and optimization via the Adam algorithm. The objective is to continue training the model until the training loss no longer shows a stable decrease.
- **Test Loss:** This represents the Cross Entropy Loss between the predicted and actual values of the evaluation set. It is used to ascertain whether the model improves on unseen data. The goal is to keep the test loss as close as possible to the training loss. Training should be halted if the gap between training and test loss begins to increase significantly.
- **Accuracy:** demonstrates the model's ability to correctly predict values. Provides insights into the model's effectiveness in a user-friendly manner.

Model Training and Evaluation

Model Training

The training process utilized a 'cuda'-enabled GPU, Nvidia 3060 (6GB mobile). Two PyTorch DataLoader instances were employed for the training and evaluation datasets, with a batch size of 256. This batch size was determined to offer the most stable and rapid processing speed

during experiments. The DataLoader for the training set additionally used weighted sampling to ensure balanced sampling across uneven data classes.

Each training iteration (epoch) included the calculation of training and test losses, accuracy measurement, backpropagation, and optimization using the Adam algorithm. The model underwent 349 epochs of training, with an average time of 23.2 seconds per epoch. The final results were as follows:

- Train loss: 1.5326
- Test loss: 1.5773
- Model accuracy for evaluation data: 58.64%

Training beyond epoch 349 didn't show any further improvement in performance metrics. The model weights in file **'my_model_weights_v3_348.pth'** in the attachment.

Evaluation

The plot charts for Train Loss, Test Loss, Model Accuracy, and Learning Rate are presented in Figure 8:

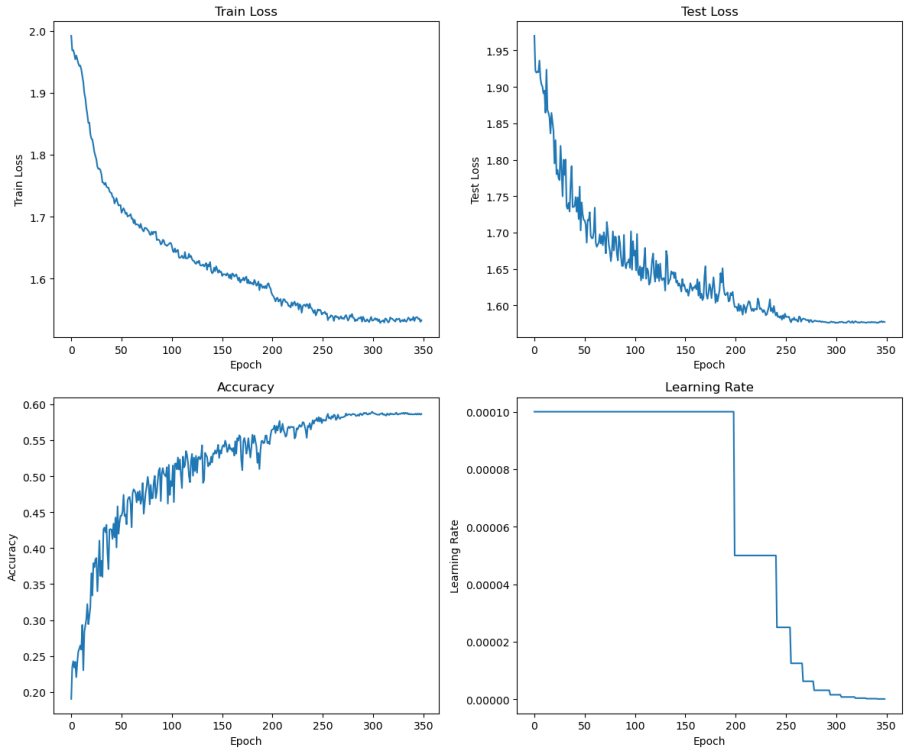
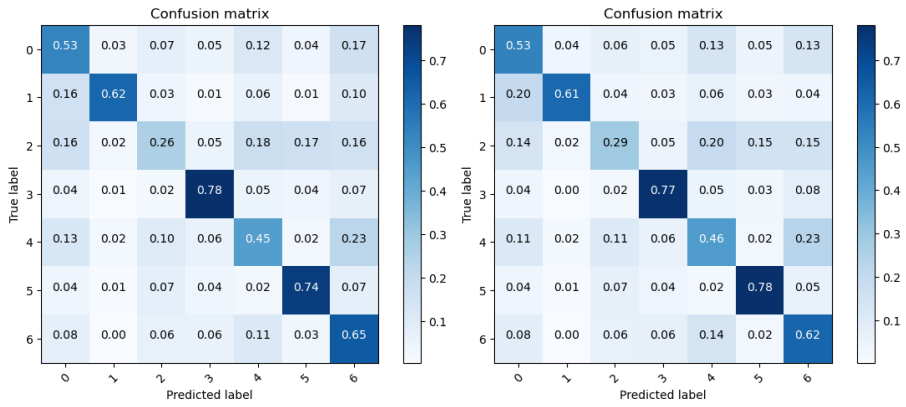


Figure 8. Model Evaluation Metrics

The analysis of the plots indicates that Test Loss and Accuracy mirror each other, and both closely follow the shape of the Train Loss curve. The model demonstrates a significant performance improvement up to epoch 50, after which it gradually declines and stabilizes beyond epoch 250. While the Train Loss curve remains smooth throughout the training process, the Test Loss exhibits more variability, stabilizing only after approximately epoch 275. The Learning Rate chart shows a constant rate of 0.0001 until epoch 200, after which it begins to decline rapidly. This decrease is due to the learning rate scheduler's attempts to reduce the learning rate in hopes of improving Test Loss. Interestingly, the impact of the learning rate

adjustments is evident at epoch 200, where Test Loss, Train Loss, and accuracy all show a noticeable improvement.

To assess the model's accuracy on unseen data, an additional **'test.csv'** subset of 7,178 samples was extracted from file **'icml_face_data.csv'** in the "Facial Expression Recognition (FER2013)" Kaggle page, which includes previously unseen public and private test samples. The accuracy and confusion matrices the test set, alongside the previously used evaluation set, are displayed in Figure 9:



Evaluation Set: Accuracy 58.62%

Test Set: Accuracy 59.00%

Figure 9. Confusion Matrixes and Accuracy Scores

The analysis of Figure 9 reveals that the model performs slightly better on the unseen Test Set, achieving an accuracy of 59%. The confusion matrices for both the Evaluation and Test sets are very similar. The model is particularly effective at recognizing samples of the '3-Happy' and '5-Surprise' classes, with very few false positives for '3-Happy'. It performs less well in recognizing '1-Disgust' and '6-Neutral' classes and shows unsatisfactory results for other categories, especially '2-Fear'. **In its current state, the model can satisfactorily classify facial expressions of the '3-Happy' and '5-Surprise' categories.**

Grad-CAM

Figure 10 shows several Grad-CAM that display "visual explanations" for decisions the model takes during categorization [3]:

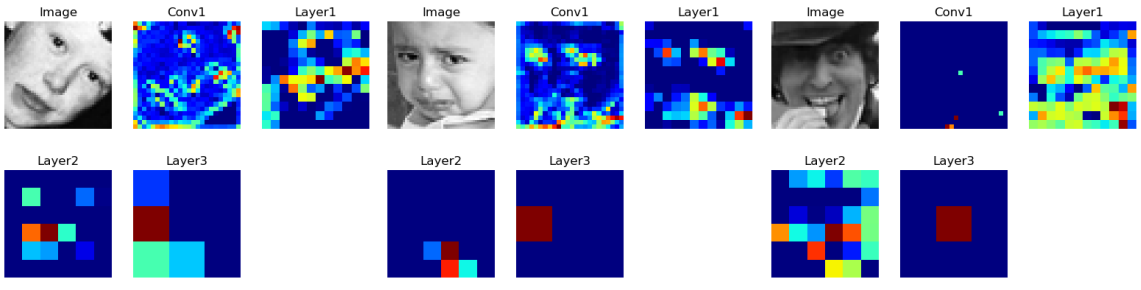


Figure 10. Grad-CAM visualisations

Conclusions

This report outlines the development of a CNN designed to recognize facial expressions from the "Facial Expression Recognition (FER2013)" dataset. It shows that deep neural networks offer rapid computation speeds and robust performance. Despite the challenge of the vanishing gradient problem, this can be effectively mitigated by employing Residual Networks.

The final model achieves an accuracy of 59.00% on the test set. While this may not be adequate for reliable recognition of all facial expressions, the model demonstrates significantly higher accuracy in identifying 'happy' (77%) and 'surprise' (78%) expressions. Consequently, it holds potential for specialized applications in recognizing these specific emotional states.

References

[1] Dumitru, Ian Goodfellow, Will Cukierski, Yoshua Bengio. (2013). Challenges in Representation Learning: Facial Expression Recognition Challenge. Kaggle. https://kaggle.com/competitions/challenges-in-representation-learning-facial-expression-recognition-challenge

[2] Kumar, T., Mileo, A., Brennan, R., & Bendechache, M. (2023). Image data augmentation approaches: A comprehensive survey and future directions. arXiv preprint arXiv:2301.02830.

[3] Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., & Batra, D. (2017). Grad-cam: Visual explanations from deep networks via gradient-based localization. In Proceedings of the IEEE international conference on computer vision (pp. 618-626).

[4] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

Appendix 1 Model Layout – text version

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 16, 24, 24]          --
|    └─Conv2d: 2-1                       [-1, 16, 48, 48]          144
|    └─BatchNorm2d: 2-2                  [-1, 16, 48, 48]          32
|    └─LeakyReLU: 2-3                    [-1, 16, 48, 48]          --
|    └─MaxPool2d: 2-4                    [-1, 16, 24, 24]          --
├─Sequential: 1-2                        [-1, 32, 12, 12]          --
|    └─ResidualBlock: 2-5                [-1, 32, 24, 24]          --
|    |    └─Sequential: 3-1              [-1, 32, 24, 24]          4,672
|    |    └─Conv2d: 3-2                  [-1, 32, 24, 24]          4,608
|    |    └─BatchNorm2d: 3-3             [-1, 32, 24, 24]          64
|    |    └─LeakyReLU: 3-4               [-1, 32, 24, 24]          --
|    |    └─Conv2d: 3-5                  [-1, 32, 24, 24]          9,216
|    |    └─BatchNorm2d: 3-6             [-1, 32, 24, 24]          64
|    |    └─LeakyReLU: 3-7               [-1, 32, 24, 24]          --
|    └─ResidualBlock: 2-6                [-1, 32, 24, 24]          --
|    |    └─Conv2d: 3-8                  [-1, 32, 24, 24]          9,216
|    |    └─BatchNorm2d: 3-9             [-1, 32, 24, 24]          64
|    |    └─LeakyReLU: 3-10              [-1, 32, 24, 24]          --
|    |    └─Conv2d: 3-11                 [-1, 32, 24, 24]          9,216
|    |    └─BatchNorm2d: 3-12            [-1, 32, 24, 24]          64
|    |    └─LeakyReLU: 3-13              [-1, 32, 24, 24]          --
|    └─MaxPool2d: 2-7                    [-1, 32, 12, 12]          --
├─Sequential: 1-3                        [-1, 64, 6, 6]            --
|    └─ResidualBlock: 2-8                [-1, 64, 12, 12]          --
|    |    └─Sequential: 3-14             [-1, 64, 12, 12]          18,560
|    |    └─Conv2d: 3-15                 [-1, 64, 12, 12]          18,432
|    |    └─BatchNorm2d: 3-16            [-1, 64, 12, 12]          128
|    |    └─LeakyReLU: 3-17              [-1, 64, 12, 12]          --
|    |    └─Conv2d: 3-18                 [-1, 64, 12, 12]          36,864
|    |    └─BatchNorm2d: 3-19            [-1, 64, 12, 12]          128
|    |    └─LeakyReLU: 3-20              [-1, 64, 12, 12]          --
|    └─ResidualBlock: 2-9                [-1, 64, 12, 12]          --
|    |    └─Conv2d: 3-21                 [-1, 64, 12, 12]          36,864
|    |    └─BatchNorm2d: 3-22            [-1, 64, 12, 12]          128
|    |    └─LeakyReLU: 3-23              [-1, 64, 12, 12]          --
|    |    └─Conv2d: 3-24                 [-1, 64, 12, 12]          36,864
|    |    └─BatchNorm2d: 3-25            [-1, 64, 12, 12]          128
|    |    └─LeakyReLU: 3-26              [-1, 64, 12, 12]          --
|    └─ResidualBlock: 2-10               [-1, 64, 12, 12]          --
|    |    └─Conv2d: 3-27                 [-1, 64, 12, 12]          36,864
|    |    └─BatchNorm2d: 3-28            [-1, 64, 12, 12]          128
|    |    └─LeakyReLU: 3-29              [-1, 64, 12, 12]          --
|    |    └─Conv2d: 3-30                 [-1, 64, 12, 12]          36,864
|    |    └─BatchNorm2d: 3-31            [-1, 64, 12, 12]          128
|    |    └─LeakyReLU: 3-32              [-1, 64, 12, 12]          --
|    └─MaxPool2d: 2-11                   [-1, 64, 6, 6]            --
├─Sequential: 1-4                        [-1, 128, 3, 3]           --
|    └─ResidualBlock: 2-12               [-1, 128, 6, 6]           --
|    |    └─Sequential: 3-33             [-1, 128, 6, 6]           73,984
|    |    └─Conv2d: 3-34                 [-1, 128, 6, 6]           73,728
|    |    └─BatchNorm2d: 3-35            [-1, 128, 6, 6]           256
|    |    └─LeakyReLU: 3-36              [-1, 128, 6, 6]           --
|    |    └─Conv2d: 3-37                 [-1, 128, 6, 6]           147,456
|    |    └─BatchNorm2d: 3-38            [-1, 128, 6, 6]           256
|    |    └─LeakyReLU: 3-39              [-1, 128, 6, 6]           --
|    └─ResidualBlock: 2-13               [-1, 128, 6, 6]           --
|    |    └─Conv2d: 3-40                 [-1, 128, 6, 6]           147,456
|    |    └─BatchNorm2d: 3-41            [-1, 128, 6, 6]           256
|    |    └─LeakyReLU: 3-42              [-1, 128, 6, 6]           --
|    |    └─Conv2d: 3-43                 [-1, 128, 6, 6]           147,456
|    |    └─BatchNorm2d: 3-44            [-1, 128, 6, 6]           256
|    |    └─LeakyReLU: 3-45              [-1, 128, 6, 6]           --
|    └─ResidualBlock: 2-14               [-1, 128, 6, 6]           --
|    |    └─Conv2d: 3-46                 [-1, 128, 6, 6]           147,456
|    |    └─BatchNorm2d: 3-47            [-1, 128, 6, 6]           256
|    |    └─LeakyReLU: 3-48              [-1, 128, 6, 6]           --
|    |    └─Conv2d: 3-49                 [-1, 128, 6, 6]           147,456
|    |    └─BatchNorm2d: 3-50            [-1, 128, 6, 6]           256
|    |    └─LeakyReLU: 3-51              [-1, 128, 6, 6]           --
|    └─ResidualBlock: 2-15               [-1, 128, 6, 6]           --
|    |    └─Conv2d: 3-52                 [-1, 128, 6, 6]           147,456
|    |    └─BatchNorm2d: 3-53            [-1, 128, 6, 6]           256
|    |    └─LeakyReLU: 3-54              [-1, 128, 6, 6]           --
|    |    └─Conv2d: 3-55                 [-1, 128, 6, 6]           147,456
|    |    └─BatchNorm2d: 3-56            [-1, 128, 6, 6]           256
|    |    └─LeakyReLU: 3-57              [-1, 128, 6, 6]           --
|    └─MaxPool2d: 2-16                   [-1, 128, 3, 3]           --
|    └─Dropout2d: 2-17                   [-1, 128, 3, 3]           --
├─Sequential: 1-5                        [-1, 2048]                --
|    └─Linear: 2-18                      [-1, 2048]                2,361,344
|    └─LeakyReLU: 2-19                   [-1, 2048]                --
|    └─Dropout: 2-20                     [-1, 2048]                --
├─Sequential: 1-6                        [-1, 7]                   --
|    └─Linear: 2-21                      [-1, 7]                   14,343
|    └─Softmax: 2-22                     [-1, 7]                   --
==========================================================================================
Total params: 3,817,079
Trainable params: 3,817,079
Non-trainable params: 0
Total mult-adds (M): 103.61
==========================================================================================
```