

ДОДАТОК А

Розрахунки обчислювальної та об'ємної складності алгоритму виділення ознак для голосового сигналу

Позначимо:

$F=78$ – кількість фрагментів для аналізу;

$L=256$ – довжина фрагмента у семплах;

$N=512$ – розмір FFT;

$M=78$ – кількість мел-фільтрів;

$K=2N+1=257$ – розмір одностороннього спектра.

Загальна кількість відліків вхідного сигналу: $F \cdot L = 78 \cdot 256 = 19968$.

Для розрідженого подання мел-фільтрів використовуємо властивість трикутного мел-фільтробанку: кожен спектральний бін бере участь не більш ніж у двох сусідніх мел-фільтрах, тобто кількість ненульових коефіцієнтів матриці фільтробанку обмежена зверху величиною $2K=514$.

Під час розрахунку не враховуються операції організації циклів (так як це залежить від конкретної реалізації), а також порівняння і операції доступу до пам'яті.

Етап 1: поділ на 78 фрагментів по 256 відліків і нормалізація.

Фрагменти копіюються у матрицю 78×256 . На цьому ж етапі доцільно організувати нормалізацію, використовуючи приведення до діапазону $[-1, 1]$, що додає 19968 операцій ділення.

Етап 2: пре-емфазис.

Згідно формули:

$$y[n] = x[n] - \alpha x[n-1], n = 1, \dots, 255 \quad (\text{A.1})$$

Для одного фрагмента: множень: 255, віднімань: 255.

Для всього сигналу: множень: $78 \cdot 255 = 19890$,
віднімань: 19890 (сумарно 39780 арифметичних операцій).

Етап 3: застосування вікна Ханна.

Згідно формули:

$$x_w[n] = x[n] \cdot w[n] \quad (\text{A.2})$$

Для одного фрагмента: множень: 256.

Для всього сигналу: множень: $78 \cdot 256 = 19968$

Етап 4: обчислення спектру.

Спочатку виконується нульове доповнення до N : для кожного фрагмента копіюються 256 наявних відліків, записуються 256 нулів. Обчислюється операція перетворення Фур'є розміру N . Розглянуто реалізацію «radix-2 butterfly» алгоритму FFT. Обчислення під час ініціалізації не враховано, так як вони виконуються один раз, а їх результат зберігається та використовується протягом усього часу виконання алгоритму.

На кожній butterfly-ітерації циклічно виконується код:

```
m := ia + N2
re_temp := c * data[2*m] + s * data[2*m+1]
im_temp := c * data[2*m + 1] - s * data[2*m]
data[2*m] := data[2*ia] - re_temp
data[2*m+1] := data[2*ia+1] - im_temp
data[2*ia] := data[2*ia] + re_temp
data[2*ia+1] := data[2*ia+1] + im_temp
```

На 1 butterfly маємо множень: 16 та додавань/віднімань: 13 (сумарно 29 операцій).

Кількість стадій для алгоритму radix-2 Cooley–Tukey визначається як:

$$S = \log_2 N = \log_2 512 = 9 \quad (\text{A.3})$$

Кількість butterfly-ітерацій на стадію:

$$N_2 = N / 2 = 512 / 2 = 256 \quad (\text{A.4})$$

На кожній стадії також виконуються наступні строки коду:

```
c := w[2*j]
s := w[2*j+1]
ia += N2
```

Отже, додатково на кожній стадії маємо додаткові 2 операції додавання та 2 операції множення (сумарно 4 операції). Загалом на одній стадії маємо арифметичних операцій: $29 \cdot 256 + 4 = 7428$.

Для всього вікна маємо операцій: $9 \cdot 7428 = 66852$.

Для всього сигналу кількість операцій становить: $78 \cdot 66852 = 5214456$.

Останньою операцією цього етапу є обчислення спектральної потужності для кожного з K бінів одностороннього спектра за формулою:

$$P[k] = \Re(X[k])^2 + \Im(X[k])^2 \quad (\text{A.5})$$

На один мел-бін: 2 множення, 1 додавання.

Для одного фрагмента: множень: $2 \cdot 257 = 514$, додавань: 257.

Для всього сигналу: множень: $78 \cdot 514 = 40092$, додавань: $78 \cdot 257 = 20046$ (сумарно – 60138 операцій).

Етап 5: застосування мел-фільтрів.

Нехай nnz_H – кількість ненульових коефіцієнтів мел-фільтробанку.

Для трикутних локальних фільтрів: $\text{nnz}_H \leq 2K = 514$.

Отже, для кожного ненульового коефіцієнта виконується: множень: 1, додавань до акумулятора відповідного мел-каналу: 1.

Верхня оцінка для одного фрагмента: множень: 514, додавань: 514.

Для всього сигналу: множень: $78 \cdot 514 = 40092$, додавань: $78 \cdot 514 = 40092$ (80184 арифметичні операції).

Етап 6: перетворення у децибели.

Згідно формул:

$$L_i = 10 \log_{10} \left(\frac{\max(P_i, a_{\min})}{\text{ref}} \right), \quad (\text{A.6})$$

$$L_i = \max(L_i, L_{\max} - \text{topdb}) \quad (\text{A.7})$$

Для одного фрагмента:

– ділення на ref: ділень: 78, записів: 78.

– обчислення десяткового логарифму: логарифмів: 78, множень: 78, (так як десятковий логарифм обчислюється на основі натурального через множення на константу).

– множення на 10: множень: 78.

– обчислення $L_{\max} - \text{topdb}$: віднімань: 1.

Для всього сигналу: ділень: $78 \cdot 78 = 6084$, логарифмів: 6084, множень: 12168, віднімань: 78 (сумарно 24414 арифметичні операції).

Етап 7: об'єднання фрагментів у фінальну матрицю ознак та нормалізація.

Нормалізація виконується за формулою:

$$z = \frac{x - \mu}{\sigma} \quad (\text{A.9})$$

Фінальна матриця ознак містить $78 \times 78 = 6084$ елементів. Якщо результат кожного фрагмента піддається перетворенню згідно формули (Б.9): маємо віднімань: 6084, ділень: 6084 (сумарно: 12168 арифметичні операції).

У підсумку, для всього сигналу маємо операцій: 5471076.

Дані, що зберігаються при ініціалізації та зберігаються протягом виконання алгоритму:

- коефіцієнти вікна Ханна: 256 коеф., тип даних: float32, обсяг: $256 \cdot 4 = 1024$ Б = 1,00 КіБ;
- twiddle-коефіцієнти: 512 значень, тип даних: float32, обсяг: $512 \cdot 4 = 2048$ Б = 2,00 КіБ;
- lookup-таблиці для бітової перестановки: 480 значень, тип даних: uint16, обсяг пам'яті: $480 \cdot 2 = 960$ Б;
- мел-фільтри (при розрідженому поданні): значення коефіцієнтів: 514 коеф., тип даних: float32, обсяг: $514 \cdot 4 = 2056$ Б $\approx 2,01$ КіБ; індекси стовпців: 514 інд., тип даних: uint16, обсяг: $514 \cdot 2 = 1028$ Б $\approx 1,00$ КіБ; вказівник рядків: 79 інд., тип даних: uint16, обсяг: $79 \cdot 2 = 158$ Б $\approx 0,154$ КіБ. Разом для мел-фільтрів: $2056 + 1028 + 158 = 3242$ Б $\approx 3,17$ КіБ.

Таблиця А.1 – Характеристики об'єму масивів, що використовуються в алгоритмі

Масив	Розмір	Тип даних	Обсяг
Вхідний сигнал	19968	int16	$39936\text{Б} = 39,0$ КіБ
Фрагментований сигнал	$78 \times 256 = 19968$	float32	$79872\text{Б} = 78,0$ КіБ
Після доповнення	$78 \times 2 \times 512 = 79872$	float32	$319488\text{Б} = 312,0$ КіБ
Потужність спектра	$78 \times 257 = 20046$	float32	$80184\text{Б} \approx 78,3$ КіБ
Мелспектрограма	$78 \times 78 = 6084$	float32	$24336\text{Б} \approx 23,8$ КіБ

Отже, сумарний обсяг пам'яті, необхідної для виконання алгоритму (за умови статичного виділення), враховуючи масиви з таблиці Б.1 та допоміжні дані, складає: $39936\text{Б} + 79872\text{Б} + 319488\text{Б} + 80184\text{Б} + 24336\text{Б} + 1024\text{Б} + 2048\text{Б} + 960\text{Б} + 3242\text{Б} = 51090\text{Б} \approx 538,2$ КіБ.

За умови динамічного виділення пам'яті на кожному етапі обробки (з вивільненням невикористовуваних даних) та збереження лише коефіцієнтів віконної функції, Twiddle, мел-фільтрів та таблиці перестановок бітів, максимальний обсяг займаної пам'яті складе: $1024\text{Б} + 2048\text{Б} + 960\text{Б} + 3242\text{Б} + 319488\text{Б} = 326762\text{Б} \approx 319$ КіБ.

ДОДАТОК Б

Програмний код

Лістинг Б.1 – Формування набору даних

```
# Combine all DataFrames into a single DataFrame
combined_df = pd.concat(
    # dataframes with filename and emotion columns
    # containing corresponding datasets
    [cremad_df, ravdess_df, surrey_df,
     toronto_df, urdu_df, emodb_df,
     resd_df, mesd_df, noise_df],
    ignore_index=True
)

# Show summary of the combined dataset
print(combined_df.head())
print("\nCombined dataset shape:", combined_df.shape)
print("\nEmotion          value          counts:\n",
combined_df['emotion'].value_counts())
# Map some emotions to 'neutral'
replace_map = {
    'calm': 'neutral',
    'enthusiasm': 'neutral',
    'boredom': 'neutral'
}

combined_df['emotion'] =
combined_df['emotion'].replace(replace_map)

# Remove 'pleasant', 'surprise' emotions from the combined
DataFrame
combined_df =
combined_df[~combined_df['emotion'].isin(['pleasant',
'surprise'])]

# Show summary of the combined dataset
print(combined_df.head())
print("\nCombined dataset shape:", combined_df.shape)
```

Продовження лістингу Б.1

```
print("\nEmotion          value          counts:\n",
combined_df['emotion'].value_counts())
```

Лістинг Б.2 – Попередня обробка набору даних

```
import os
import pandas as pd
import librosa
import soundfile as sf
import numpy as np
import webrtcvad
from tqdm import tqdm

output_dir = "F:/speech_emotions_dataset"
os.makedirs(output_dir, exist_ok=True)
target_sr = 8000

segment_length = 2.5    # Target segment length, seconds
min_remainder = 1.0     # Minimum length of the final segment
to keep, seconds

vad_mode = 2            # Aggressive VAD mode
frame_duration_ms = 30  # Frame size for VAD (10, 20, or 30
ms)

max_silence_ms = 200    # max allowed silence gap to merge
into speech

min_voiced_ms = 300     # minimum length of voiced region to
keep

global_idx = 0
vad = webrtcvad.Vad(vad_mode)
frame_size = int(target_sr * frame_duration_ms / 1000)
max_silence_frames = max_silence_ms // frame_duration_ms
min_voiced_frames = min_voiced_ms // frame_duration_ms
new_rows = []

for _, row in tqdm(combined_df.iterrows(),
total=len(combined_df)):
    filename = row["filename"]
    emotion = row["emotion"]
```

Продовження лістингу Б.2

```

try:
    y, sr = librosa.load(filename, sr=target_sr,
mono=True)
except Exception as e:
    print(f"Failed to load {filename}: {e}")
    continue
# WebRTC VAD with gap merging (skip for "noise" class)
if emotion != "noise":
    y_pcm = (y * 32768).astype(np.int16)
    num_frames = len(y_pcm) // frame_size
    speech_flags = []
    for i in range(num_frames):
        start = i * frame_size
        end = start + frame_size
        frame = y_pcm[start:end]
        if len(frame) < frame_size:
            speech_flags.append(False)
            continue
        speech_flags.append(vad.is_speech(frame.tobyte
s(), target_sr))
    # Gap merging
    merged_flags = speech_flags.copy()
    silence_count = 0
    for i in range(len(speech_flags)):
        if not speech_flags[i]:
            silence_count += 1
        else:
            if 0 < silence_count <= max_silence_frames:
                # merge small silence gap
                for j in range(i - silence_count, i):
                    merged_flags[j] = True
            silence_count = 0
    # Extract voiced regions
    voiced_segments = []

```

Продовження лістингу Б.2

```

start_frame = None
voiced_count = 0
for i, flag in enumerate(merged_flags):
    if flag:
        if start_frame is None:
            start_frame = i
            voiced_count += 1
    else:
        if start_frame is not None:
            if voiced_count >= min_voiced_frames:
                s = start_frame * frame_size
                e = i * frame_size
                voiced_segments.append(y_pcm[s:e])
            start_frame = None
            voiced_count = 0
# Handle trailing speech segment
if start_frame is not None and voiced_count >=
min_voiced_frames:
    s = start_frame * frame_size
    e = len(y_pcm)
    voiced_segments.append(y_pcm[s:e])
if len(voiced_segments) == 0:
    continue
# Concatenate voiced segments
y_pcm = np.concatenate(voiced_segments)
y = y_pcm.astype(np.float32) / 32768.0
# Segmentation
samples_per_segment = int(segment_length * target_sr)
min_remainder_samples = int(min_remainder * target_sr)
total_samples = len(y)
if total_samples == 0:
    continue
num_segments = total_samples // samples_per_segment
for i in range(num_segments):

```

```

        start = i * samples_per_segment
        end = start + samples_per_segment
        segment = y[start:end]
        new_filename = f"sample_{global_idx}.wav"
        sf.write(os.path.join(output_dir, new_filename),
segment, target_sr)
        new_rows.append([new_filename, emotion])
        global_idx += 1
        remainder = total_samples - num_segments *
samples_per_segment
        if remainder >= min_remainder_samples:
            start = num_segments * samples_per_segment
            segment = y[start:]
            new_filename = f"sample_{global_idx}.wav"
            sf.write(os.path.join(output_dir, new_filename),
segment, target_sr)
            new_rows.append([new_filename, emotion])
            global_idx += 1
    new_metadata_df = pd.DataFrame(new_rows,
columns=["filename", "emotion"])
    print(new_metadata_df.head())
    print("\nCombined dataset shape:", new_metadata_df.shape)
    print("\nEmotion          value          counts:\n",
new_metadata_df['emotion'].value_counts())
    new_metadata_df.to_csv(os.path.join(output_dir,
"metadata.csv"), index=False)

```

Лістинг Б.3 – Виділення ознак

```

class AudioToSpec(torch.nn.Module):
    def __init__(self, sample_rate=8000, target_len=2.5,
n_mels=78,
                    n_fft=512, hop_length=256, augment=True):
        super().__init__()
        self.sample_rate = sample_rate

```

Продовження лістингу Б.3

```

        self.target_len = int(sample_rate * target_len /
hop_length)

    # Pre-emphasis
    self.preemph = T.Preemphasis(coeff=0.97)
    # Spectrogram pipeline
    self.spectrogram = T.Spectrogram(n_fft=n_fft,
                                     hop_length=hop_length,
                                     power=None,
                                     center=False)
    self.melscale = T.MelScale(n_mels=n_mels,
                              sample_rate=sample_rate,
                              n_stft=n_fft//2+1)
    self.db = T.AmplitudeToDB(stype="power", top_db=80)
    self.augment = augment
    if self.augment:
        # SpecAugment pipeline
        self.stretch =
T.TimeStretch(hop_length=hop_length, n_freq=n_fft//2+1)
        self.freq_mask =
T.FrequencyMasking(freq_mask_param=20)
        self.time_mask =
T.TimeMasking(time_mask_param=5)
    def pad_or_trim_spec(self, spec):
        _, time = spec.shape
        if time > self.target_len:
            spec = spec[:, :self.target_len]
        elif time < self.target_len:
            pad_time = self.target_len - time
            spec = torch.nn.functional.pad(spec, (0,
pad_time, 0, 0))
        return spec
    def forward(self, waveform):
        # Pre-emphasis

```

Продовження лістингу Б.3

```

        waveform = self.preemph(waveform)
        # Spectrogram
        spec = self.spectrogram(waveform)
        if self.augment:
            # SpecAugment
            rate = np.random.uniform(0.8, 1.2)
            spec = self.stretch(spec, overriding_rate=rate)

        power_spec = spec.abs().pow(2)

        if self.augment:
            # SpecAugment
            power_spec = self.freq_mask(power_spec)
            power_spec = self.time_mask(power_spec)
            # Pad/trim spectrogram to target shape
            power_spec = self.pad_or_trim_spec(power_spec.squeeze(0))

        # Spectrogram → MelScale → dB
        mel = self.melscale(power_spec)
        mel_db = self.db(mel)

        if self.augment:
            noise = torch.randn_like(mel_db)
            mel_db += noise

        return mel_db

class EmotionDataset(Dataset):
    def __init__(self, dataset_path, sample_rate=8000,
target_len=2.5,
n_mels=78, n_fft=512, hop_length=256,
augment=True):
        self.audio_dir = dataset_path

```

Продовження лістингу Б.3

```

        self.df = pd.read_csv(os.path.join(self.audio_dir,
"metadata.csv"))
        self.label_map = {label: idx for idx, label in
enumerate(
            sorted(self.df['emotion'].unique())
        )}
        self.processor = AudioToSpec(sample_rate,
target_len, n_mels,
                                     n_fft, hop_length,
augment)
    def __len__(self):
        return len(self.df)
    def __getitem__(self, idx):
        row = self.df.iloc[idx]
        filepath = os.path.join(self.audio_dir,
row['filename'])
        waveform, sr = torchaudio.load(filepath)
        spec = self.processor(waveform).unsqueeze(0)
        label = self.label_map[row["emotion"]]
        return spec, label

```

Лістинг Б.4 – Стратифікований поділ на навчальну та тестову вибірки з балансуванням

```

def stratified_split(dataset, test_fraction=0.2, seed=42):
    labels =
dataset.df['emotion'].map(dataset.label_map).to_numpy()
    rng = np.random.RandomState(seed)
    indices = np.arange(len(labels))
    train_idx = []
    test_idx = []
    for cls in np.unique(labels):
        cls_indices = indices[labels == cls]
        rng.shuffle(cls_indices)
        n_test = int(len(cls_indices) * test_fraction)

```

Продовження лістингу Б.4

```

        if test_fraction > 0 and n_test == 0:
            n_test = 1
            test_sel = cls_indices[:n_test]
            train_sel = cls_indices[n_test:]
            test_idx.extend(test_sel.tolist())
            train_idx.extend(train_sel.tolist())
        rng.shuffle(train_idx)
        rng.shuffle(test_idx)
        return train_idx, test_idx
full_dataset = EmotionDataset(
    DATASET_PATH,
    sample_rate=SAMPLERATE,
    target_len=FIXED_SECONDS,
    n_mels=N_MELS,
    n_fft=N_FFT,
    hop_length=HOP_LENGTH
)
features_root = os.path.join(DATASET_PATH, "features")
os.makedirs(features_root, exist_ok=True)
os.makedirs(os.path.join(features_root, "train"),
exist_ok=True)
os.makedirs(os.path.join(features_root, "test"),
exist_ok=True)
label_map = full_dataset.label_map
with open(os.path.join(features_root, "label_map.json"),
"w") as f:
    json.dump(label_map, f, indent=4)
n_classes = len(label_map)
train_idx, test_idx = stratified_split(full_dataset,
TEST_SUBSET_SIZE, seed=42)
train_ds = Subset(full_dataset, train_idx)
test_ds = Subset(full_dataset, test_idx)
print(f"Full dataset size: {len(full_dataset)} samples")
print(f"Train dataset size: {len(train_ds)} samples")

```

Продовження лістингу Б.4

```

print(f"Test dataset size: {len(test_ds)} samples")
# Build class-balanced sampler for training
train_labels = [full_dataset.df.iloc[i]['emotion'] for i in
train_idx]
train_label_ids = np.array([full_dataset.label_map[l] for
l in train_labels])
# Build class-balanced sampler for testing
test_labels = [full_dataset.df.iloc[i]['emotion'] for i in
test_idx]
test_label_ids = np.array([full_dataset.label_map[l] for l
in test_labels])
train_class_counts      =      np.bincount(train_label_ids,
minlength=n_classes)
test_class_counts       =      np.bincount(test_label_ids,
minlength=n_classes)
print("Train:", train_class_counts)
print("Test:", test_class_counts)
generator = torch.Generator().manual_seed(42)
train_sampler = WeightedRandomSampler(
    1.0 / train_class_counts[train_label_ids],
    num_samples=len(train_label_ids),
    replacement=True,
    generator=generator
)
test_sampler = WeightedRandomSampler(
    1.0 / test_class_counts[test_label_ids],
    num_samples=len(test_label_ids),
    replacement=True,
    generator=generator
)

```

Лістинг Б.5 – Побудова моделі «Vanilla CNN»

```

import torch
import torch.nn as nn
class AudioVanillaCNN(nn.Module):
    def __init__(
        self,
        num_classes,
        depth=3,
        base_channels=16,
        width_mult=1.0,
        use_batchnorm=False,
        use_stride_conv=False
    ):
        super().__init__()
        layers = []
        in_channels = 1
        out_channels = int(base_channels * width_mult)
        for _ in range(depth):
            stride = 2 if use_stride_conv else 1
            layers.append(
                nn.Conv2d(
                    in_channels,
                    out_channels,
                    kernel_size=3,
                    stride=stride,
                    padding=1,
                    bias=not use_batchnorm
                )
            )
            if use_batchnorm:
                layers.append(nn.BatchNorm2d(out_channels))
            layers.append(nn.ReLU(inplace=True))
            if not use_stride_conv:
                layers.append(nn.MaxPool2d(2))
            in_channels = out_channels

```

Продовження лістингу Б.5

```

        out_channels = int(out_channels * 2)

    self.features = nn.Sequential(*layers)

    self.global_pool = nn.AdaptiveAvgPool2d(1)
    self.classifier = nn.Linear(in_channels,
num_classes)

    def forward(self, x):
        x = self.features(x)
        x = self.global_pool(x)
        x = torch.flatten(x, 1)
        return self.classifier(x)

```

Лістинг Б.6 – Побудова моделі «MobileNet V1* та V2*»

```

import torch
import torch.nn as nn
import torch.nn.functional as F
import math

def _make_divisible(v, divisor=8):
    return int(math.ceil(v / divisor) * divisor)

class DepthwiseSeparableConv(nn.Module):
    def __init__(self, in_channels, out_channels, stride):
        super().__init__()

        self.depthwise = nn.Sequential(
            nn.Conv2d(in_channels, in_channels, 3, stride,
padding=1, groups=in_channels,
bias=False),

            nn.BatchNorm2d(in_channels),
            nn.ReLU(inplace=True)
        )

        self.pointwise = nn.Sequential(

```

Продовження лістингу Б.6

```

        nn.Conv2d(in_channels, out_channels, 1,
bias=False),

        nn.BatchNorm2d(out_channels),
        nn.ReLU(inplace=True)
    )
    def forward(self, x):
        x = self.depthwise(x)
        x = self.pointwise(x)
        return x

class AudioMobileNetV1(nn.Module):
    def __init__(self, num_classes, width_mult=1.0,
depth_mult=1.0):
        super().__init__()
        alpha = width_mult
        rho = depth_mult
        def c(ch):
            return _make_divisible(ch * alpha)
        input_channel = c(32)
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, input_channel, 3, stride=2,
padding=1, bias=False),
            nn.BatchNorm2d(input_channel),
            nn.ReLU(inplace=True)
        )
        # (out_channels, stride, repeats)
        cfg = [
            (64, 1, 1),
            (128, 2, 2),
            (256, 2, 2),
            (512, 2, 6),
            (1024, 2, 2),
        ]
        layers = []
        for out_c, stride, repeats in cfg:

```

Продовження лістингу Б.6

```

        out_channel = c(out_c)
        repeats = max(1, int(math.ceil(repeats * rho)))
        for i in range(repeats):
            s = stride if i == 0 else 1
            layers.append(
                DepthwiseSeparableConv(input_channel,
out_channel, s)
            )
            input_channel = out_channel
        self.features = nn.Sequential(*layers)
        self.global_pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Linear(input_channel, num_classes)
    def forward(self, x):
        x = self.conv1(x)
        x = self.features(x)
        x = self.global_pool(x)
        x = torch.flatten(x, 1)
        return self.fc(x)

class InvertedResidual(nn.Module):
    def __init__(self, in_channels, out_channels, stride,
expand_ratio):
        super().__init__()
        hidden_dim = int(round(in_channels * expand_ratio))
        self.use_res_connect = stride == 1 and in_channels
== out_channels
        layers = []
        if expand_ratio != 1:
            layers.extend([
                nn.Conv2d(in_channels, hidden_dim, 1,
bias=False),
                nn.BatchNorm2d(hidden_dim),
                nn.ReLU6(inplace=True)
            ])
        layers.extend([

```

Продовження лістингу Б.6

```

        nn.Conv2d(hidden_dim, hidden_dim, 3, stride,
                  padding=1, groups=hidden_dim,
bias=False),

        nn.BatchNorm2d(hidden_dim),
        nn.ReLU6(inplace=True),
        nn.Conv2d(hidden_dim, out_channels, 1,
bias=False),

        nn.BatchNorm2d(out_channels),
    ])
    self.conv = nn.Sequential(*layers)
    def forward(self, x):
        if self.use_res_connect:
            return x + self.conv(x)
        return self.conv(x)
class AudioMobileNetV2(nn.Module):
    def __init__(self, num_classes, width_mult=1.0,
depth_mult=1.0):
        super().__init__()
        alpha = width_mult
        rho = depth_mult
        def c(ch):
            return _make_divisible(ch * alpha)
        input_channel = c(32)
        last_channel = 1280 if alpha <= 1.0 else c(1280)
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, input_channel, 3, stride=2,
padding=1, bias=False),
            nn.BatchNorm2d(input_channel),
            nn.ReLU6(inplace=True)
        )
        # t, c, n, s
        inverted_residual_setting = [
            (1, 16, 1, 1),
            (6, 24, 2, 2),

```

Продовження лістингу Б.6

```

        (6, 32, 3, 2),
        (6, 64, 4, 2),
        (6, 96, 3, 1),
        (6, 160, 3, 2),
        (6, 320, 1, 1),
    ]
    layers = []
    for t, c_out, n, s in inverted_residual_setting:
        out_channel = c(c_out)
        repeats = max(1, int(math.ceil(n * rho)))
        for i in range(repeats):
            stride = s if i == 0 else 1
            layers.append(
                InvertedResidual(input_channel,
out_channel, stride, t)
            )
            input_channel = out_channel
    self.blocks = nn.Sequential(*layers)
    self.conv_last = nn.Sequential(
        nn.Conv2d(input_channel, last_channel, 1,
bias=False),
        nn.BatchNorm2d(last_channel),
        nn.ReLU6(inplace=True)
    )
    self.global_pool = nn.AdaptiveAvgPool2d(1)
    self.fc = nn.Linear(last_channel, num_classes)
    def forward(self, x):
        x = self.conv1(x)
        x = self.blocks(x)
        x = self.conv_last(x)
        x = self.global_pool(x)
        x = torch.flatten(x, 1)
        return self.fc(x)

```

Лістинг Б.7 – Побудова моделі «DenseNet*»

```

import torch
import torch.nn as nn
import torch.nn.functional as F

class DenseLayer(nn.Module):
    def __init__(self, in_channels, growth_rate,
bn_size=4):
        super().__init__()
        inter_channels = bn_size * growth_rate
        self.layer = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels, inter_channels,
kernel_size=1, bias=False),
            nn.BatchNorm2d(inter_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(inter_channels, growth_rate,
kernel_size=3, padding=1, bias=False)
        )
    def forward(self, x):
        new_features = self.layer(x)
        return new_features

class DenseBlock(nn.Module):
    def __init__(self, in_channels, growth_rate,
num_layers, bn_size=4):
        super().__init__()
        self.layers = nn.ModuleList()
        channels = in_channels
        for _ in range(num_layers):
            self.layers.append(DenseLayer(channels,
growth_rate, bn_size))
            channels += growth_rate
        self.out_channels = channels
    def forward(self, x):
        features = [x]

```

Продовження лістингу Б.7

```

        for layer in self.layers:
            new_feat = layer(torch.cat(features, dim=1))
            features.append(new_feat)
        return torch.cat(features, dim=1)

class TransitionLayer(nn.Module):
    def __init__(self, in_channels, compression=0.5):
        super().__init__()
        out_channels = int(in_channels * compression)
        self.layer = nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels, out_channels,
kernel_size=1, bias=False),
            nn.AvgPool2d(kernel_size=2, stride=2)
        )
        self.out_channels = out_channels
    def forward(self, x):
        return self.layer(x)

class AudioDenseNet(nn.Module):
    def __init__(
        self,
        num_classes,
        growth_rate=16,
        block_layers=(4, 4, 4),
        initial_channels=32,
        compression=0.5,
        bn_size=4
    ):
        super().__init__()
        self.stem = nn.Conv2d(
            1,
            initial_channels,
            kernel_size=3,
            padding=1,

```

Продовження лістингу Б.7

```

        bias=False
    )
    channels = initial_channels
    self.blocks = nn.ModuleList()
    self.transitions = nn.ModuleList()
    for i, num_layers in enumerate(block_layers):
        block = DenseBlock(
            channels,
            growth_rate,
            num_layers,
            bn_size
        )
        self.blocks.append(block)
        channels = block.out_channels
        if i != len(block_layers) - 1:
            trans = TransitionLayer(
                channels,
                compression=compression
            )
            self.transitions.append(trans)
            channels = trans.out_channels
    self.final_bn = nn.BatchNorm2d(channels)
    self.classifier = nn.Linear(channels, num_classes)
    self.final_channels = channels
def forward(self, x):
    x = self.stem(x)
    for i, block in enumerate(self.blocks):
        x = block(x)
        if i < len(self.transitions):
            x = self.transitions[i](x)
    x = F.relu(self.final_bn(x))
    x = F.adaptive_avg_pool2d(x, (1, 1))
    x = torch.flatten(x, 1)
    return self.classifier(x)

```

Лістинг Б.8 – Побудова моделі «ResNet*»

```

import torch
import torch.nn as nn
import torch.nn.functional as F
def _make_divisible(v, divisor=8):
    return int((v + divisor - 1) // divisor * divisor)
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels,
stride=1):
    super().__init__()
    self.conv1 = nn.Conv2d(
        in_channels, out_channels,
        kernel_size=3, stride=stride,
        padding=1, bias=False
    )
    self.bn1 = nn.BatchNorm2d(out_channels)
    self.conv2 = nn.Conv2d(
        out_channels, out_channels,
        kernel_size=3, stride=1,
        padding=1, bias=False
    )
    self.bn2 = nn.BatchNorm2d(out_channels)
    self.shortcut = nn.Identity()
    if stride != 1 or in_channels != out_channels:
        self.shortcut = nn.Sequential(
            nn.Conv2d(
                in_channels, out_channels,
                kernel_size=1, stride=stride,
                bias=False
            ),
            nn.BatchNorm2d(out_channels)
        )
    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))

```

```

        out += self.shortcut(x)
        return F.relu(out)

class AudioResNet(nn.Module):
    def __init__(
        self,
        num_classes,
        depth=(1, 1, 1),
        width_mult=1.0,
        base_channels=(16, 32, 64)
    ):
        super().__init__()
        c1 = _make_divisible(base_channels[0] * width_mult)
        c2 = _make_divisible(base_channels[1] * width_mult)
        c3 = _make_divisible(base_channels[2] * width_mult)
        self.conv1 = nn.Conv2d(
            1, c1, kernel_size=3,
            stride=1, padding=1,
            bias=False
        )
        self.bn1 = nn.BatchNorm2d(c1)
        self.layer1 = self._make_layer(c1, c1, depth[0],
stride=1)
        self.layer2 = self._make_layer(c1, c2, depth[1],
stride=2)
        self.layer3 = self._make_layer(c2, c3, depth[2],
stride=2)

        self.global_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(c3, num_classes)
        def _make_layer(self, in_channels, out_channels,
blocks, stride):
            layers = []
            layers.append(ResidualBlock(in_channels,
out_channels, stride))
            for _ in range(1, blocks):
                layers.append(ResidualBlock(out_channels,
out_channels))

```

Продовження лістингу Б.8

```

        return nn.Sequential(*layers)
    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.global_pool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)
        return x

```

Лістинг Б.9– Побудова моделі «ShuffleNet V1* та V2*»

```

import torch
import torch.nn as nn
import torch.nn.functional as F
def make_divisible(v, divisor=8):
    return int((v + divisor - 1) // divisor * divisor)
class ChannelShuffle(nn.Module):
    def __init__(self, groups):
        super().__init__()
        self.groups = groups
    def forward(self, x):
        B, C, H, W = x.size()
        g = self.groups
        x = x.view(B, g, C // g, H, W)
        x = x.transpose(1, 2).contiguous()
        return x.view(B, C, H, W)
class ShuffleNetV1Block(nn.Module):
    def __init__(self, in_channels, out_channels, stride,
groups):
        super().__init__()
        self.stride = stride
        self.groups = groups
        bottleneck_channels = out_channels // 4

```

Продовження лістингу Б.9

```

    if stride == 2:
        out_channels -= in_channels
    # 1x1 group conv
    self.conv1 = nn.Conv2d(
        in_channels,
        bottleneck_channels,
        1,
        groups=groups,
        bias=False
    )
    self.bn1 = nn.BatchNorm2d(bottleneck_channels)
    self.shuffle = ChannelShuffle(groups)
    self.dwconv = nn.Conv2d( # depthwise 3x3
        bottleneck_channels,
        bottleneck_channels,
        3,
        stride=stride,
        padding=1,
        groups=bottleneck_channels,
        bias=False
    )
    self.bn2 = nn.BatchNorm2d(bottleneck_channels)
    self.conv2 = nn.Conv2d( # 1x1 group conv
        bottleneck_channels,
        out_channels,
        1,
        groups=groups,
        bias=False
    )
    self.bn3 = nn.BatchNorm2d(out_channels)
    if stride == 2:
        self.shortcut = nn.AvgPool2d(
            kernel_size=3,
            stride=2,

```

Продовження лістингу Б.9

```

        padding=1
    )
    else:
        self.shortcut = nn.Identity()
def forward(self, x):
    out = F.relu(self.bn1(self.conv1(x)))
    out = self.shuffle(out)
    out = self.bn2(self.dwconv(out))
    out = self.bn3(self.conv2(out))
    if self.stride == 2:
        return F.relu(torch.cat([self.shortcut(x), out],
dim=1))
    else:
        return F.relu(x + out)
class AudioShuffleNetV1(nn.Module):
    def __init__(
        self,
        num_classes,
        groups=3,
        widths=(240, 480, 960),
        depths=(4, 8, 4)):
        super().__init__()
        widths = [make_divisible(w) for w in widths]
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 24, 3, stride=2, padding=1,
bias=False),
            nn.BatchNorm2d(24),
            nn.ReLU(inplace=True)
        )
        self.stage2 = self._make_stage(
            24, widths[0], depths[0], groups
        )
        self.stage3 = self._make_stage(
            widths[0], widths[1], depths[1], groups

```

Продовження лістингу Б.9

```

    )
    self.stage4 = self._make_stage(
        widths[1], widths[2], depths[2], groups
    )
    self.global_pool = nn.AdaptiveAvgPool2d((1, 1))
    self.fc = nn.Linear(widths[2], num_classes)
def _make_stage(self, in_c, out_c, blocks, groups):
    layers = []
    layers.append(
        ShuffleNetV1Block(in_c, out_c, stride=2,
groups=groups)
    )
    for _ in range(1, blocks):
        layers.append(
            ShuffleNetV1Block(out_c, out_c, stride=1,
groups=groups)
        )
    return nn.Sequential(*layers)
def forward(self, x):
    x = self.conv1(x)
    x = self.stage2(x)
    x = self.stage3(x)
    x = self.stage4(x)
    x = self.global_pool(x)
    x = torch.flatten(x, 1)
    return self.fc(x)
class ShuffleNetV2Block(nn.Module):
    def __init__(self, in_channels, out_channels, stride):
        super().__init__()
        self.stride = stride
        branch_channels = out_channels // 2
        if stride == 1:
            assert in_channels == out_channels
        self.shuffle = ChannelShuffle(2)

```

Продовження лістингу Б.9

```

        # Branch2 (always exists)
        self.branch2 = nn.Sequential(
            nn.Conv2d(
                in_channels if stride > 1 else
branch_channels,
                branch_channels,
                1,
                bias=False
            ),
            nn.BatchNorm2d(branch_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(
                branch_channels,
                branch_channels,
                3,
                stride=stride,
                padding=1,
                groups=branch_channels,
                bias=False
            ),
            nn.BatchNorm2d(branch_channels),
            nn.Conv2d(
                branch_channels,
                branch_channels,
                1,
                bias=False
            ),
            nn.BatchNorm2d(branch_channels),
            nn.ReLU(inplace=True),
        )
    if stride == 2:
        self.branch1 = nn.Sequential(
            nn.Conv2d(
                in_channels,

```

Продовження лістингу Б.9

```

        in_channels,
        3,
        stride=2,
        padding=1,
        groups=in_channels,
        bias=False
    ),
    nn.BatchNorm2d(in_channels),
    nn.Conv2d(
        in_channels,
        branch_channels,
        1,
        bias=False
    ),
    nn.BatchNorm2d(branch_channels),
    nn.ReLU(inplace=True),
)

def forward(self, x):
    if self.stride == 1:
        x1, x2 = torch.chunk(x, 2, dim=1)
        out = torch.cat((x1, self.branch2(x2)), dim=1)
    else:
        out = torch.cat((self.branch1(x),
self.branch2(x)), dim=1)
    return self.shuffle(out)

class AudioShuffleNetV2(nn.Module):
    def __init__(
        self,
        num_classes,
        depths=(4, 8, 4),
        base_channels=(116, 232, 464)
    ):
        super().__init__()
        c1 = 24

```

Продовження лістингу Б.9

```

        c2 = make_divisible(base_channels[0])
        c3 = make_divisible(base_channels[1])
        c4 = make_divisible(base_channels[2])
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, c1, 3, stride=2, padding=1,
bias=False),
            nn.BatchNorm2d(c1),
            nn.ReLU(inplace=True)
        )
        self.stage2 = self._make_stage(c1, c2, depths[0])
        self.stage3 = self._make_stage(c2, c3, depths[1])
        self.stage4 = self._make_stage(c3, c4, depths[2])
        self.global_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(c4, num_classes)
    def _make_stage(self, in_c, out_c, blocks):
        layers = []
        layers.append(ShuffleNetV2Block(in_c, out_c,
stride=2))
        for _ in range(1, blocks):
            layers.append(ShuffleNetV2Block(out_c, out_c,
stride=1))
        return nn.Sequential(*layers)
    def forward(self, x):
        x = self.conv1(x)
        x = self.stage2(x)
        x = self.stage3(x)
        x = self.stage4(x)
        x = self.global_pool(x)
        x = torch.flatten(x, 1)
        return self.fc(x)

```

Лістинг Б.10 – Побудова моделі «SqueezeNet*»

```

import torch
import torch.nn as nn
import torch.nn.functional as F
def make_divisible(v, divisor=8):
    return int((v + divisor - 1) // divisor * divisor)
class Fire(nn.Module):
    def __init__(self, in_channels, squeeze_channels,
expand_channels):
        super().__init__()
        self.squeeze = nn.Sequential(
            nn.Conv2d(in_channels, squeeze_channels, 1,
bias=False),
            nn.BatchNorm2d(squeeze_channels),
            nn.ReLU(inplace=True)
        )
        self.expand1x1 = nn.Sequential(
            nn.Conv2d(squeeze_channels, expand_channels, 1,
bias=False),
            nn.BatchNorm2d(expand_channels),
            nn.ReLU(inplace=True)
        )
        self.expand3x3 = nn.Sequential(
            nn.Conv2d(squeeze_channels, expand_channels, 3,
padding=1, bias=False),
            nn.BatchNorm2d(expand_channels),
            nn.ReLU(inplace=True)
        )
    def forward(self, x):
        x = self.squeeze(x)
        return torch.cat([
            self.expand1x1(x),
            self.expand3x3(x)
        ], dim=1)
class AudioSqueezeNet(nn.Module):

```

Продовження лістингу Б.10

```

def __init__(
    self,
    num_classes,
    width_mult=1.0,
    depth=1,
    squeeze_ratio=0.25,
    base_channels=64
):
    super().__init__()
    c1 = make_divisible(32 * width_mult)
    c2 = make_divisible(base_channels * width_mult)
    self.conv1 = nn.Sequential(
        nn.Conv2d(1, c1, 3, stride=2, padding=1,
bias=False),
        nn.BatchNorm2d(c1),
        nn.ReLU(inplace=True)
    )
    self.maxpool1 = nn.MaxPool2d(2, stride=2)
    self.fire_blocks = nn.Sequential(
        *self._make_fire_stack(c1, c2, depth,
squeeze_ratio)
    )
    self.global_pool = nn.AdaptiveAvgPool2d((1, 1))
    self.fc = nn.Linear(c2, num_classes)
    def _make_fire_stack(self, in_c, out_c, depth,
squeeze_ratio):
        layers = []
        for i in range(depth):
            squeeze_c = make_divisible(out_c *
squeeze_ratio)
            layers.append(Fire(in_c, squeeze_c, out_c //
2))
        in_c = out_c
    return layers

```

Продовження лістингу Б.10

```
def forward(self, x):
    x = self.conv1(x)
    x = self.maxpool1(x)
    x = self.fire_blocks(x)
    x = self.global_pool(x)
    x = torch.flatten(x, 1)
    return self.fc(x)
```

Лістинг Б.11 – Побудова моделі «SE-Net*»

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SEBlock(nn.Module):
    def __init__(self, channels, reduction=16):
        super().__init__()
        self.fc1 = nn.Linear(channels, channels //
reduction, bias=False)
        self.fc2 = nn.Linear(channels // reduction,
channels, bias=False)
    def forward(self, x):
        # x: (B, C, H, W)
        b, c, _, _ = x.size()
        y = F.adaptive_avg_pool2d(x, 1).view(b, c) #
squeeze
        y = F.relu(self.fc1(y))
        y = torch.sigmoid(self.fc2(y))
        y = y.view(b, c, 1, 1)
        return x * y # scale

class SEConvBlock(nn.Module):
    def __init__(self, in_ch, out_ch, kernel_size=3,
stride=1, padding=1, reduction=16):
        super().__init__()
        self.conv = nn.Conv2d(in_ch, out_ch, kernel_size,
stride, padding, bias=False)
```

Продовження лістингу Б.11

```

        self.bn = nn.BatchNorm2d(out_ch)
        self.se = SEBlock(out_ch, reduction)
        self.relu = nn.ReLU(inplace=True)
    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        x = self.se(x)
        x = self.relu(x)
        return x

class AudioSENet(nn.Module):
    def __init__(self, num_classes=6, base_channels=32,
num_blocks=4):
        super().__init__()
        layers = []
        in_channels = 1
        ch = base_channels
        for _ in range(num_blocks):
            layers.append(SEConvBlock(in_channels, ch))
            layers.append(nn.MaxPool2d(2))
            in_channels = ch
            ch *= 2
        self.features = nn.Sequential(*layers)
        self.global_pool = nn.AdaptiveAvgPool2d(1)
        self.classifier = nn.Linear(in_channels,
num_classes)

    def forward(self, x):
        x = self.features(x)
        x = self.global_pool(x) # (B, C, 1, 1)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x

```

Лістинг Б.12 – Побудова моделі «RepVGG*»

```

import torch
import torch.nn as nn
import torch.nn.functional as F
def make_divisible(v, divisor=8):
    return int((v + divisor - 1) // divisor * divisor)
class RepVGGBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1,
deploy=False):
        super().__init__()
        self.deploy = deploy
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.stride = stride
        if deploy:
            self.rbr_reparam = nn.Conv2d(
                in_channels, out_channels, 3,
                stride=stride, padding=1, bias=True
            )
        else:
            self.rbr_dense = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, 3,
stride, 1, bias=False),
                nn.BatchNorm2d(out_channels)
            )
            self.rbr_1x1 = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, 1,
stride, 0, bias=False),
                nn.BatchNorm2d(out_channels)
            )
            if in_channels == out_channels and stride == 1:
                self.rbr_identity =
nn.BatchNorm2d(in_channels)
            else:
                self.rbr_identity = None

```

Продовження лістингу Б.12

```

def forward(self, x):
    if self.deploy:
        return F.relu(self.rbr_reparam(x))
    out = self.rbr_dense(x) + self.rbr_1x1(x)
    if self.rbr_identity is not None:
        out += self.rbr_identity(x)
    return F.relu(out)

def get_equivalent_kernel_bias(self):
    k3, b3 = self._fuse_branch(self.rbr_dense)
    k1, b1 = self._fuse_branch(self.rbr_1x1)
    k1 = self._pad_1x1_to_3x3(k1)
    if self.rbr_identity is not None:
        kid, bid = self._fuse_identity()
    else:
        kid, bid = 0, 0
    return k3 + k1 + kid, b3 + b1 + bid

def _fuse_branch(self, branch):
    if branch is None:
        return 0, 0
    conv, bn = branch[0], branch[1]
    w = conv.weight
    mean = bn.running_mean
    var = bn.running_var
    gamma = bn.weight
    beta = bn.bias
    eps = bn.eps
    std = torch.sqrt(var + eps)
    t = (gamma / std).reshape(-1, 1, 1, 1)
    return w * t, beta - mean * gamma / std

def _fuse_identity(self):
    kernel = torch.zeros(
        (self.out_channels, self.in_channels, 3, 3),
        device=self.rbr_identity.weight.device
    )

```

Продовження лістингу Б.12

```

        for i in range(self.in_channels):
            kernel[i, i, 1, 1] = 1.0
        bn = self.rbr_identity
        mean = bn.running_mean
        var = bn.running_var
        gamma = bn.weight
        beta = bn.bias
        eps = bn.eps
        std = torch.sqrt(var + eps)
        t = (gamma / std).reshape(-1, 1, 1, 1)
        return kernel * t, beta - mean * gamma / std

    @staticmethod
    def _pad_1x1_to_3x3(kernel):
        if isinstance(kernel, int):
            return 0
        padded = torch.zeros(
            (kernel.size(0), kernel.size(1), 3, 3),
            device=kernel.device
        )
        padded[:, :, 1:2, 1:2] = kernel
        return padded

    def switch_to_deploy(self):
        if self.deploy:
            return
        kernel, bias = self.get_equivalent_kernel_bias()
        self.rbr_reparam = nn.Conv2d(
            self.in_channels,
            self.out_channels,
            3,
            stride=self.stride,
            padding=1,
            bias=True
        )
        self.rbr_reparam.weight.data = kernel

```

Продовження лістингу Б.12

```

        self.rbr_reparam.bias.data = bias
        del self.rbr_dense
        del self.rbr_1x1
        if hasattr(self, "rbr_identity"):
            del self.rbr_identity
        self.deploy = True
class AudioRepVGG(nn.Module):
    def __init__(
        self,
        num_classes,
        depth=1,
        base_channels=(64, 128, 256),
        deploy=False
    ):
        super().__init__()
        c0 = 32
        c1 = make_divisible(base_channels[0])
        c2 = make_divisible(base_channels[1])
        c3 = make_divisible(base_channels[2])
        self.stage0 = nn.Sequential(
            nn.Conv2d(1, c0, 3, stride=2, padding=1,
bias=False),
            nn.BatchNorm2d(c0),
            nn.ReLU(inplace=True)
        )
        self.stage1 = self._make_stage(c0, c1, depth,
stride=2, deploy=deploy)
        self.stage2 = self._make_stage(c1, c2, depth,
stride=2, deploy=deploy)
        self.stage3 = self._make_stage(c2, c3, depth,
stride=2, deploy=deploy)
        self.global_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(c3, num_classes)

```

Продовження лістингу Б.12

```

def _make_stage(self, in_c, out_c, num_blocks, stride,
deploy):
    blocks = []
    blocks.append(RepVGGBlock(in_c, out_c,
stride=stride, deploy=deploy))
    for _ in range(1, num_blocks):
        blocks.append(RepVGGBlock(out_c, out_c,
stride=1, deploy=deploy))
    return nn.Sequential(*blocks)

def forward(self, x):
    x = self.stage0(x)
    x = self.stage1(x)
    x = self.stage2(x)
    x = self.stage3(x)
    x = self.global_pool(x)
    x = torch.flatten(x, 1)
    return self.fc(x)

def switch_to_deploy(self):
    for module in self.modules():
        if isinstance(module, RepVGGBlock):
            module.switch_to_deploy()

```

Лістинг Б.13 – Побудова моделі «ConvMixer*»

```

import torch
import torch.nn as nn
import torch.nn.functional as F
class ConvMixerBlock(nn.Module):
    def __init__(self, dim, kernel_size):
        super().__init__()
        self.dwconv = nn.Conv2d(dim, dim, kernel_size,
                                groups=dim,
                                padding=kernel_size//2)
        self.bn1 = nn.BatchNorm2d(dim)
        self.pwconv = nn.Conv2d(dim, dim, kernel_size=1)

```

Продовження лістингу Б.13

```

        self.bn2 = nn.BatchNorm2d(dim)
        self.act = nn.GELU()
    def forward(self, x):
        x = x + self.act(self.bn1(self.dwconv(x)))
        x = self.act(self.bn2(self.pwconv(x)))
        return x

class AudioConvMixer(nn.Module):
    def __init__(self, num_classes, dim=64, depth=4,
patch_size=2, kernel_size=5):
        super(AudioConvMixer, self).__init__()
        # Patch embedding
        self.patch_embed = nn.Conv2d(1, dim,
kernel_size=patch_size, stride=patch_size)
        self.bn = nn.BatchNorm2d(dim)
        # ConvMixer blocks
        self.blocks = nn.Sequential(
            *[ConvMixerBlock(dim, kernel_size) for _ in
range(depth)]
        )
        # Classification head
        self.global_pool = nn.AdaptiveAvgPool2d((1,1))
        self.fc = nn.Linear(dim, num_classes)
    def forward(self, x):
        x = F.relu(self.bn(self.patch_embed(x)))
        x = self.blocks(x)
        x = self.global_pool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)
        return x

```

Лістинг Б.14 – Тренувальний цикл

```

device = torch.device("cuda" if torch.cuda.is_available()
else "cpu")
train_loader, val_loader = get_dataloaders(DATASET_PATH,
batch_size=BATCH_SIZE, device=device)
num_classes = train_loader.dataset.num_classes
sample_input, sample_target = next(iter(val_loader))
print("Sample input shape:", sample_input.shape, "Sample
target shape:", sample_target.shape)
model = get_model(MODEL, num_classes)
print(f"Using model: {MODEL}")
model = model.to(device)
criterion =
nn.CrossEntropyLoss(label_smoothing=LABEL_SMOOTHING)
if OPTIMIZER.lower() == "adam":
    opt = optim.Adam # type: ignore
elif OPTIMIZER.lower() == "sgd":
    opt = optim.SGD # type: ignore
elif OPTIMIZER.lower() == "rmsprop":
    opt = optim.RMSprop # type: ignore
else:
    raise ValueError(f"Unsupported optimizer:
{OPTIMIZER}")
optimizer = opt(
    model.parameters(),
    lr=LR,
    weight_decay=WEIGHT_DECAY
)
# Scheduler: reduce LR on plateau
scheduler = ReduceLROnPlateau(
    optimizer,
    mode="min",
    factor=LR_SCHEDULER_FACTOR,
    patience=LR_SCHEDULER_PATIENCE
)

```

Продовження лістингу Б.14

```

# Early stopping
best_macro_f1 = -float('inf')
patience_counter = 0

# Metrics

train_acc_metric =
torchmetrics.Accuracy(task='multiclass',
num_classes=num_classes).to(device)
    val_f1_macro = torchmetrics.F1Score(task='multiclass',
num_classes=num_classes, average='macro').to(device)
    val_f1_weighted = torchmetrics.F1Score(task='multiclass',
num_classes=num_classes, average='weighted').to(device)
    val_acc_macro = torchmetrics.Accuracy(task='multiclass',
num_classes=num_classes, average='macro').to(device)
    val_acc_weighted =
torchmetrics.Accuracy(task='multiclass',
num_classes=num_classes, average='weighted').to(device)
    val_recall_macro = torchmetrics.Recall(task='multiclass',
num_classes=num_classes, average='macro').to(device)
    val_precision_macro =
torchmetrics.Precision(task='multiclass',
num_classes=num_classes, average='macro').to(device)
    val_confmat =
torchmetrics.ConfusionMatrix(task='multiclass',
num_classes=num_classes).to(device)

# TensorBoard writer
log_dir =
f"logs/{MODEL}_{datetime.now().strftime('%Y%m%d_%H%M%S')}"
writer = SummaryWriter(log_dir=log_dir, flush_secs=10)
for epoch in range(train_loader.dataset.num_epochs):
    model.train()
    train_loss, cnt = 0, 0
    train_acc_metric.reset()
    train_loader.dataset.load_epoch(epoch)
    for inputs, targets in train_loader:

```

Продовження лістингу Б.14

```

        inputs = scale_data(inputs)
        inputs = inputs.to(device)
        targets = targets.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * inputs.size(0)
        cnt += inputs.size(0)
        preds = outputs.argmax(1)
        train_acc_metric.update(preds, targets)
train_loss /= cnt
train_acc = float(train_acc_metric.compute().item())
train_acc_metric.reset()
# Validation
model.eval()
val_loss, cnt = 0, 0
val_f1_macro.reset()
val_f1_weighted.reset()
with torch.no_grad():
    for inputs, targets in val_loader:
        inputs = scale_data(inputs)
        inputs = inputs.to(device)
        targets = targets.to(device)
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        val_loss += loss.item() * inputs.size(0)
        cnt += inputs.size(0)
        preds = outputs.argmax(1)
        val_f1_macro.update(preds, targets)
        val_f1_weighted.update(preds, targets)
        val_acc_macro.update(preds, targets)
        val_acc_weighted.update(preds, targets)

```

Продовження лістингу Б.14

```

        val_recall_macro.update(preds, targets)
        val_precision_macro.update(preds, targets)
        val_confmat.update(preds, targets)
    val_loss /= cnt
    macro_f1 = float(val_f1_macro.compute().item())
    weighted_f1 = float(val_f1_weighted.compute().item())
    val_acc_macro_v =
float(val_acc_macro.compute().item())
    val_acc = val_acc_macro_v
    val_acc_weighted_v =
float(val_acc_weighted.compute().item())
    val_recall_macro_v =
float(val_recall_macro.compute().item())
    val_precision_macro_v =
float(val_precision_macro.compute().item())
    cm = val_confmat.compute().cpu().numpy()
    val_f1_macro.reset()
    val_f1_weighted.reset()
    val_acc_macro.reset()
    val_acc_weighted.reset()
    val_recall_macro.reset()
    val_precision_macro.reset()
    val_confmat.reset()
    # Scheduler
    scheduler.step(val_loss)
    # Early stopping (monitors Macro F1)
    if macro_f1 > best_macro_f1 + 1e-4:
        best_macro_f1 = macro_f1
        patience_counter = 0
        torch.save(model.state_dict(),
f'trained_models/{MODEL}.pth')
    else:
        patience_counter += 1
    lr = optimizer.param_groups[0]['lr']

```

Продовження лістингу Б.14

```

        print(f"Epoch {epoch+1}/{NUM_EPOCHS} lr: {lr:.5f} - "
              f"Train Loss: {train_loss:.4f}, Train Acc:
{train_acc:.4f}, Val Loss: {val_loss:.4f}, Val Acc:
{val_acc:.4f}, Macro F1: {macro_f1:.4f}, Weighted F1:
{weighted_f1:.4f}")
        writer.add_scalar('Train/Loss', train_loss, epoch)
        writer.add_scalar('Train/Accuracy', train_acc, epoch)
        writer.add_scalar('Val/Loss', val_loss, epoch)
        writer.add_scalar('Val/Accuracy', val_acc, epoch)
        writer.add_scalar('Val/Macro_F1', macro_f1, epoch)
        writer.add_scalar('Val/Weighted_F1', weighted_f1,
epoch)

        writer.add_scalar('Val/Accuracy_Macro',
val_acc_macro_v, epoch)
        writer.add_scalar('Val/Accuracy_Weighted',
val_acc_weighted_v, epoch)
        writer.add_scalar('Val/Recall_Macro',
val_recall_macro_v, epoch)
        writer.add_scalar('Val/Precision_Macro',
val_precision_macro_v, epoch)
        writer.add_scalar('Train/LR', lr, epoch)
        if patience_counter >= EARLY_STOPPING_PATIENCE:
            print("Early stopping: no improvement.")
            break
        # Confusion matrix
        try:
            fig, ax = plt.subplots(figsize=(6, 6))
            im = ax.imshow(cm, interpolation='nearest',
cmap='Blues')
            ax.figure.colorbar(im, ax=ax)
            ax.set_xlabel('Predicted')
            ax.set_ylabel('True')
            ax.set_title('Confusion Matrix')
            # annotate

```

Продовження лістингу Б.14

```

        if cm.size > 0:
            thresh = cm.max() / 2.0
            for i in range(cm.shape[0]):
                for j in range(cm.shape[1]):
                    ax.text(j, i, format(int(cm[i, j]),
'd'),
                                ha="center", va="center",
                                color="white" if cm[i, j] >
thresh else "black")
            writer.add_figure('Val/ConfusionMatrix', fig,
epoch)

            plt.close(fig)
        except Exception:
            pass

    try:
        writer.close()
    except Exception:
        pass

```

Лістинг Б.15 – Визначення кількості параметрів, об'ємної та часової складності моделі

```

device = torch.device("cuda") if torch.cuda.is_available()
else torch.device("cpu")
train_loader, test_loader = get_dataloaders(
    DATASET_PATH,
    batch_size=64,
    device=device
)
num_classes = test_loader.dataset.num_classes
model = get_model(MODEL, num_classes)
model.load_state_dict(torch.load(os.path.join("trained_models", f"{MODEL}.pth"), weights_only=True))
model.eval()
if hasattr(model, "switch_to_deploy"):

```

Продовження лістингу Б.15

```

        model.switch_to_deploy()
    sample_input, sample_target = next(iter(test_loader))
    print("Sample input shape:", sample_input.shape, "Sample
target shape:", sample_target.shape)
    print(f"Using model: {MODEL}")
    num_params = sum(p.numel() for p in model.parameters())
    num_trainable_params = sum(p.numel() for p in
model.parameters() if p.requires_grad)
    print("Model parameters - Total:", num_params / 1e6, "M,
Trainable: ", num_trainable_params / 1e6, "M")
    input_size = (1, *sample_input.shape[1:])
    rand_inp = torch.randn(input_size)
    flop_counter = FlopCounterMode(display=False, depth=None)
# type: ignore
    with flop_counter:
        model(rand_inp)
    flops = flop_counter.get_total_flops()
    print(f"Model FLOPs (inference): {flops / 1e6:.2f} MFLOPs")
    param_size = sum(p.numel() * p.element_size() for p in
model.parameters())
    buffer_size = sum(b.numel() * b.element_size() for b in
model.buffers())
    model_size = (param_size + buffer_size) / 1024**2 # MB

    print(f"Model size (static): {model_size:.2f} MB")
    with open(f"test_results/{MODEL}_metrics.txt", "w") as f:
        f.write(f"Model FLOPs (inference): {flops / 1e6:.2f}
MFLOPs\n")
        f.write(f"Model parameters: {num_params / 1e6:.2f}
M\n")
        f.write(f"Model size (static): {model_size:.2f} MB\n")

```

Лістинг Б.16 – Квантування моделі

```

model = get_model(MODEL, num_classes)
model.load_state_dict(torch.load(f'trained_models/{MODEL}.pth'))
model.eval()
if hasattr(model, "switch_to_deploy"):
    model.switch_to_deploy()
x, _ = next(iter(test_loader))
x_shape = [1]+list(x.shape)[1:]

quant_setting =
QuantizationSettingFactory.espdn_setting()

dispatch_table = {layer:
get_target_platform(target=TARGET, num_of_bits=bits) for layer,
bits in \

zip(HIGH_PRECISION_LAYERS,
[HIGH_PRECISION_LAYERS_BITS] * len(HIGH_PRECISION_LAYERS))}
quant_ppq_graph = espdn_quantize_torch(
    model=model,
    espdn_export_file=f"trained_models/quantized_models/{MODEL}.espdn",
    calib_dataloader=test_loader,
    calib_steps=32, # Number of calibration steps
    input_shape=x_shape, # Input shape, batchsize = 1
    target=TARGET, # Quantization target type
    num_of_bits=QUANTIZE_BITS, # Number of quantization
bits

    collate_fn=collate_fn, # Custom collate function to
extract inputs from dataloader
    settings=quant_setting, # Quantization settings
    device=DEVICE, # Device for calibration (use "cpu"
for quantization-aware training)
    error_report=True, # Enable error reporting to
identify quantization issues
    skip_export=False, # Whether to skip exporting the
quantized model (useful for debugging)

```

Продовження лістингу Б.16

```
        export_test_values=False, # Whether to export test
values for debugging quantization results
        dispatching_override=dispatch_table, # Override
quantization settings for specific layers
        verbose=1, # Output detailed log information
    )
```

ДОДАТОК В

Архітектура моделей ШНМ

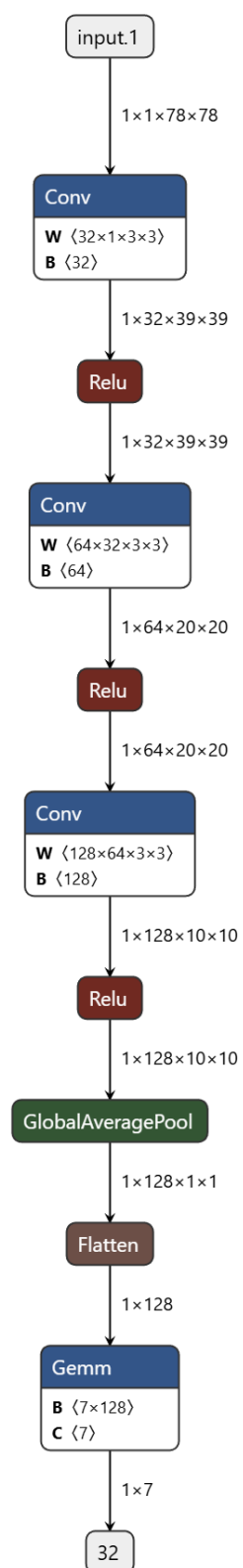


Рисунок В.1 – Архітектура «Vanilla CNN»

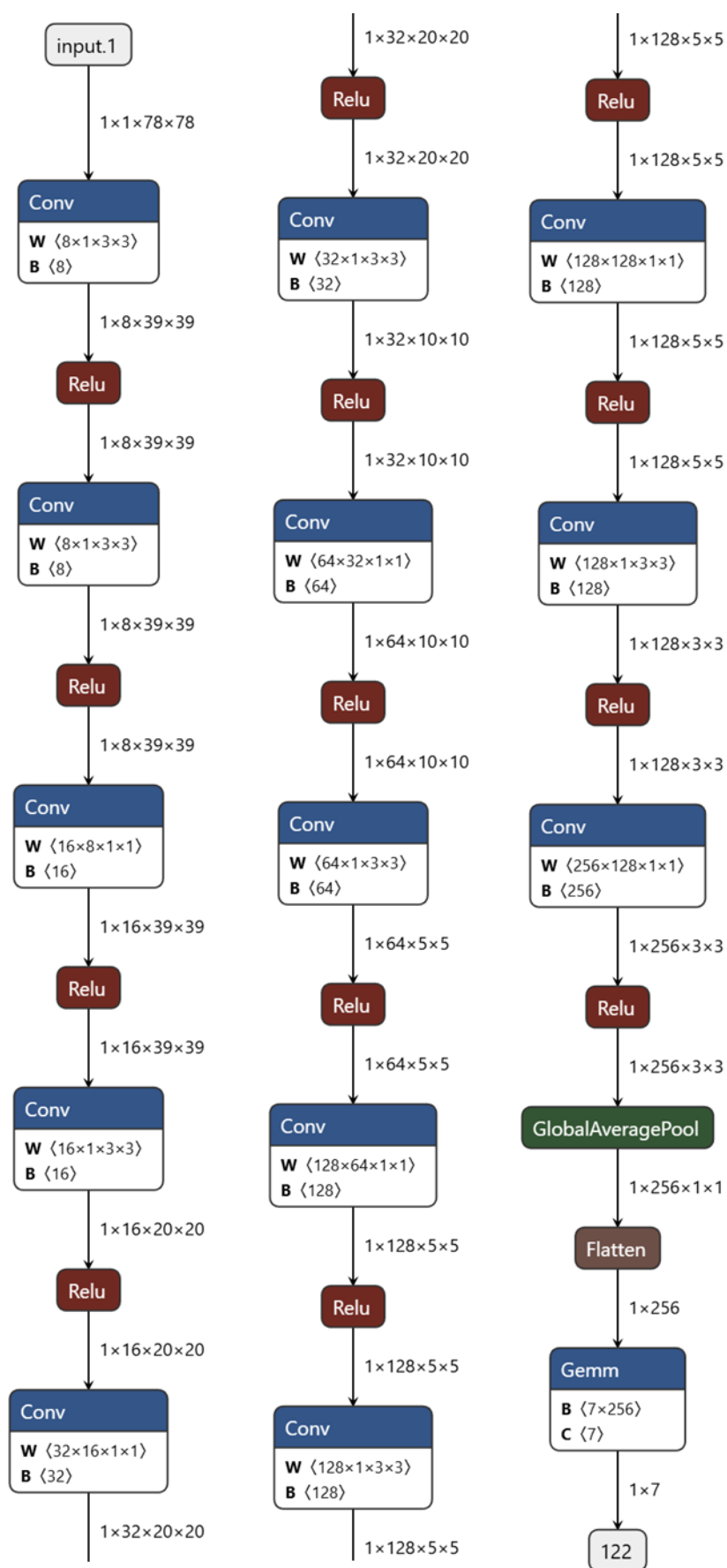


Рисунок В.2 – Архітектура «MobileNet V1*»

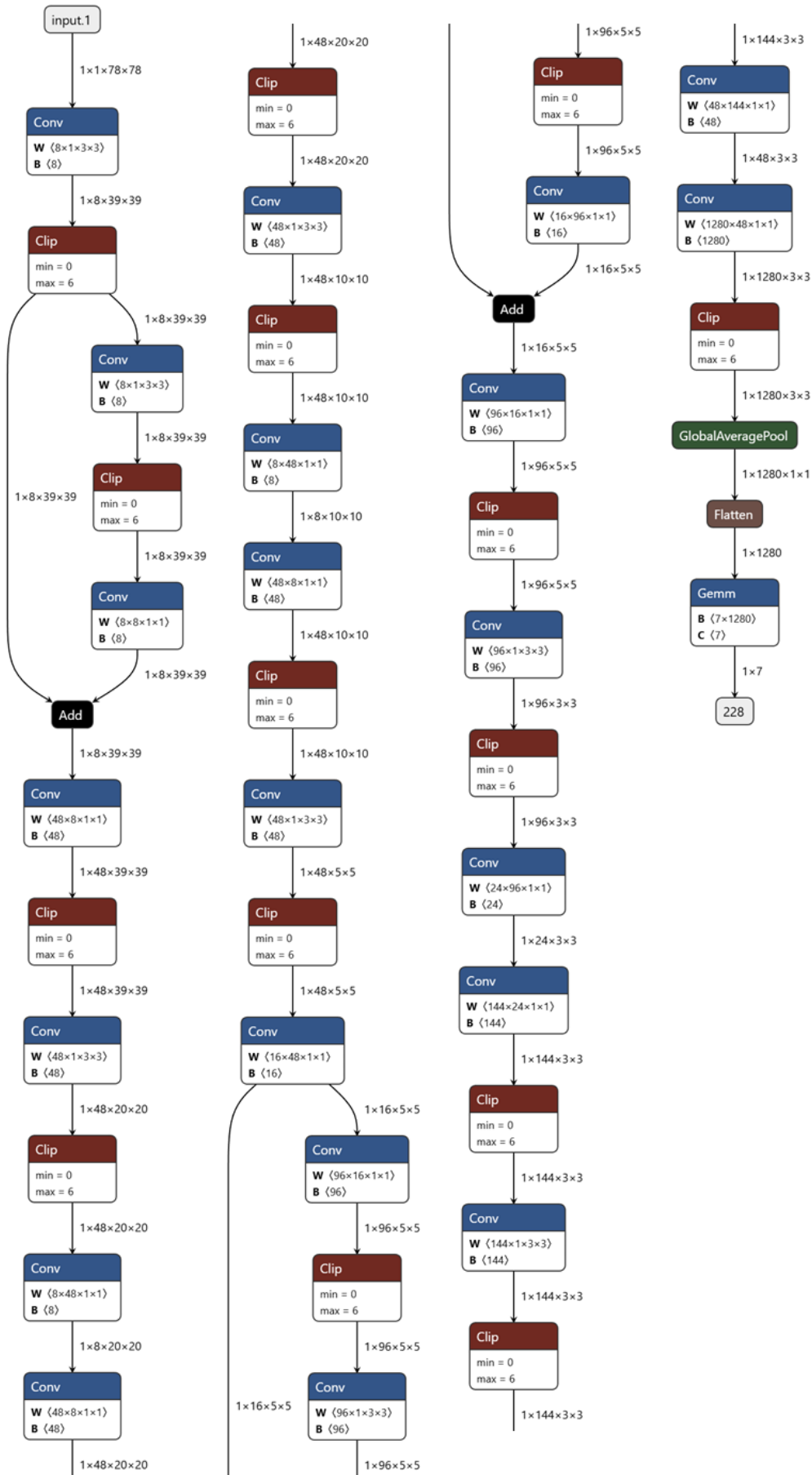


Рисунок В.3 – Архітектура «MobileNet V2*»

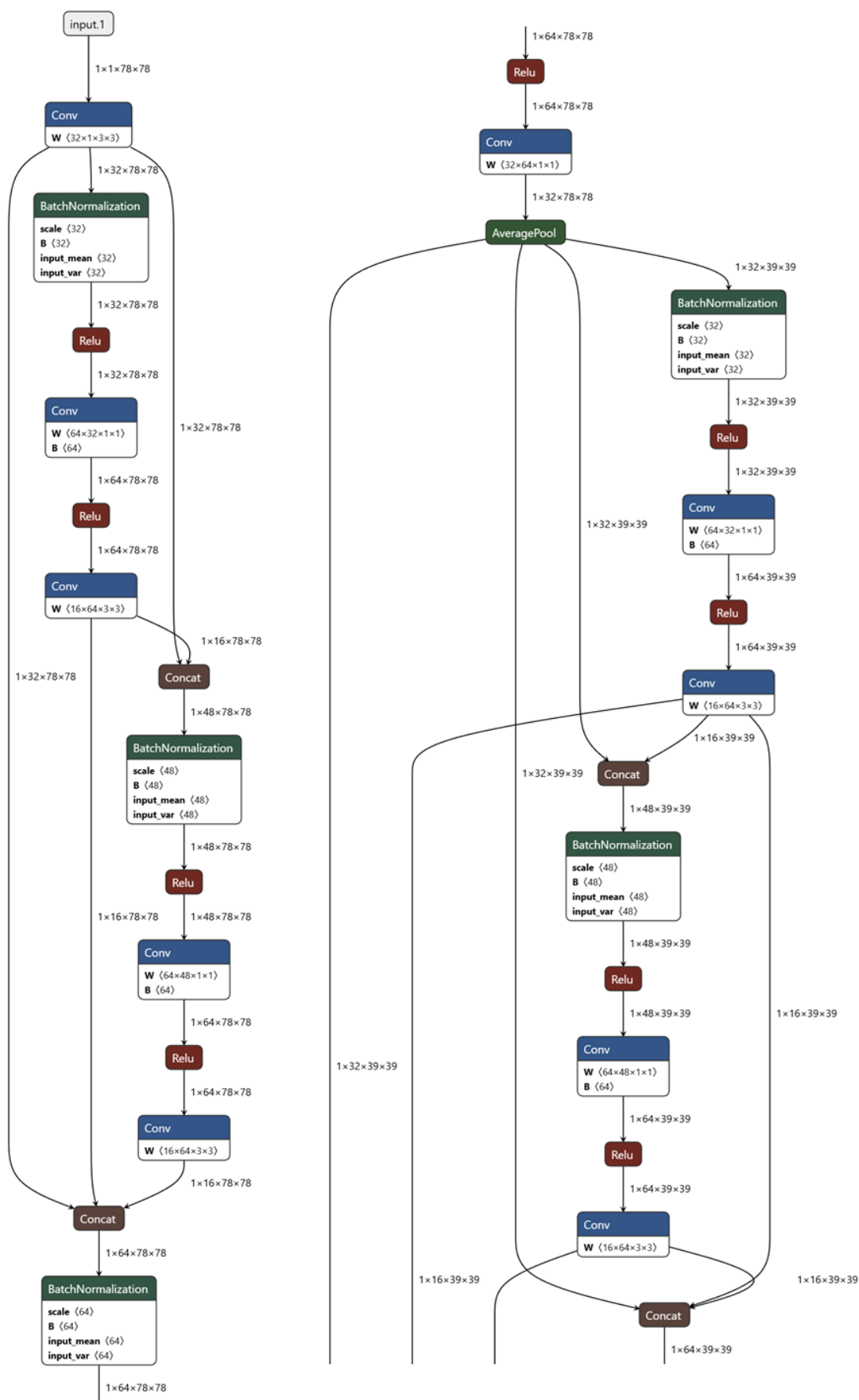


Рисунок В.4 – Архітектура «DenseNet*»

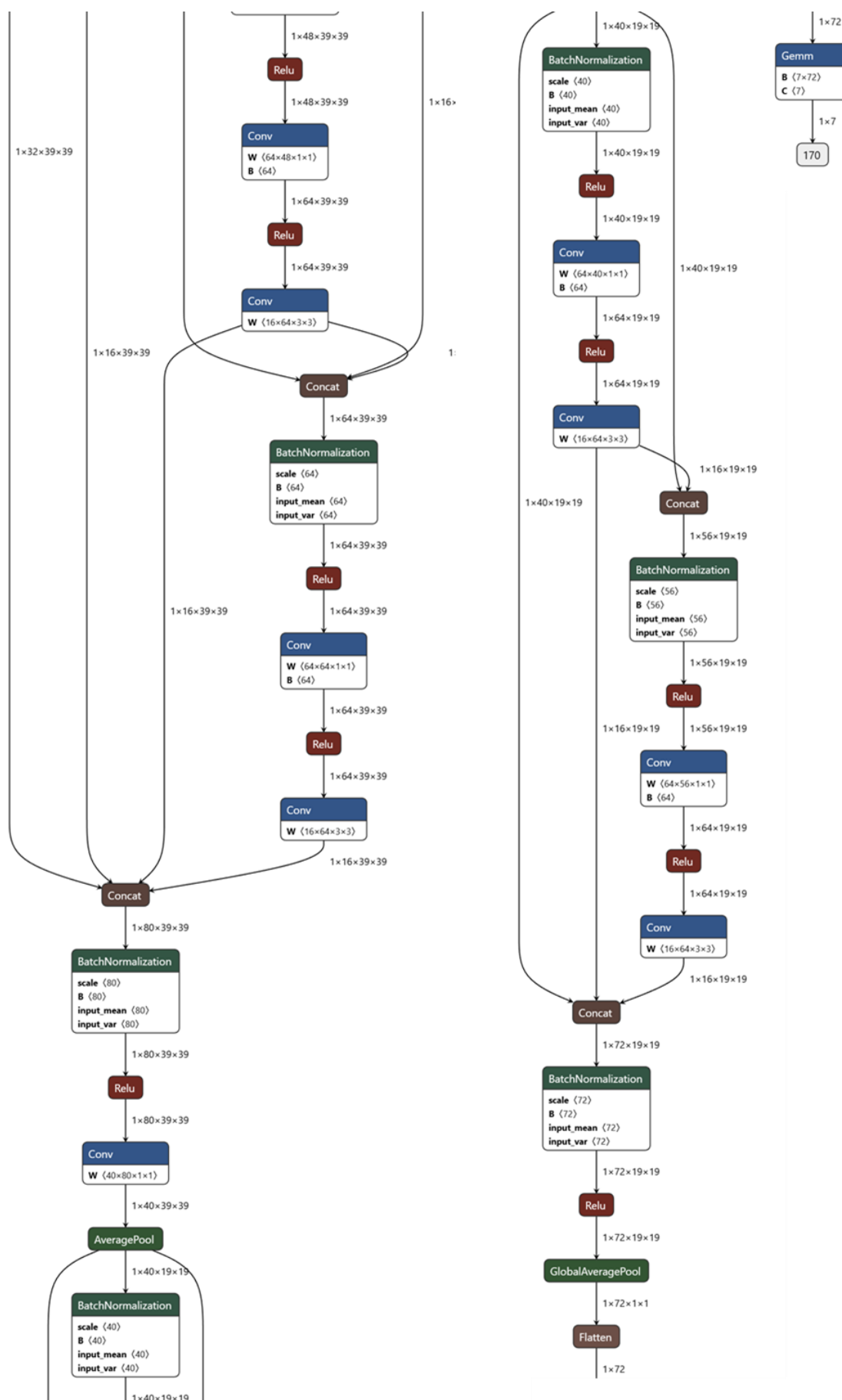


Рисунок В.4 – Архітектура «DenseNet*» (продовження)

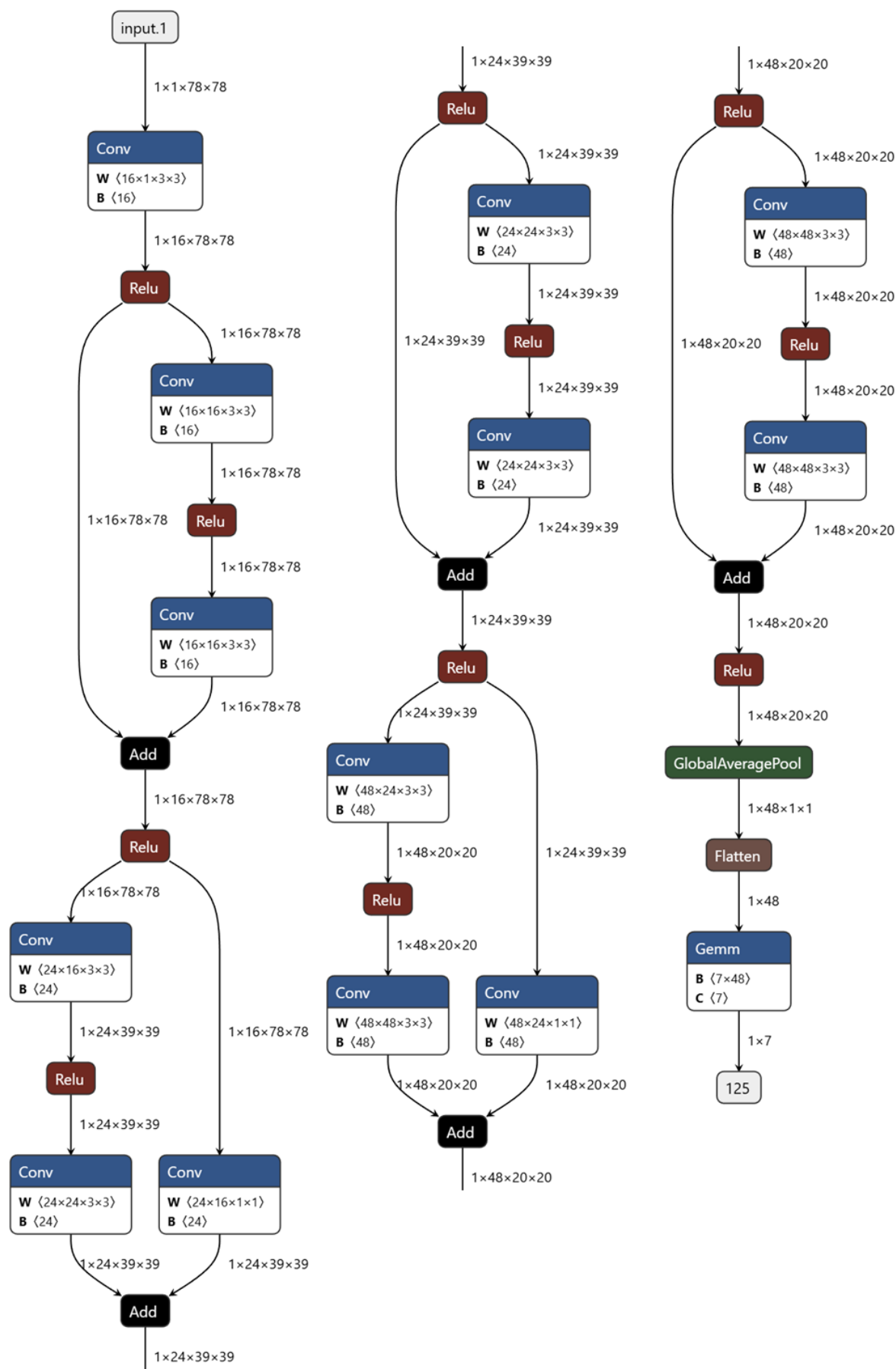


Рисунок В.5 – Архітектура «ResNet*»

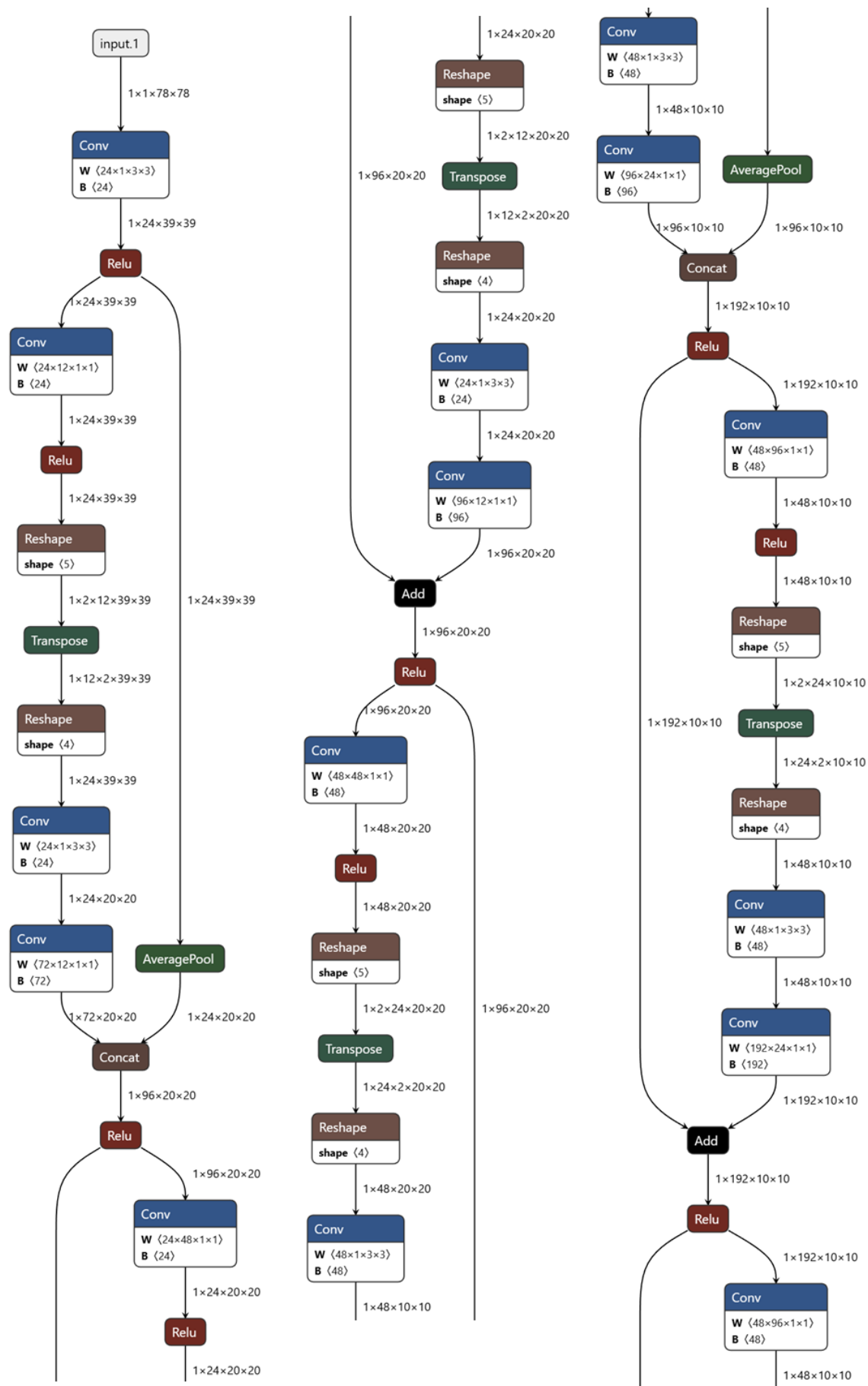


Рисунок В.6 – Архітектура «ShuffleNet V1*»

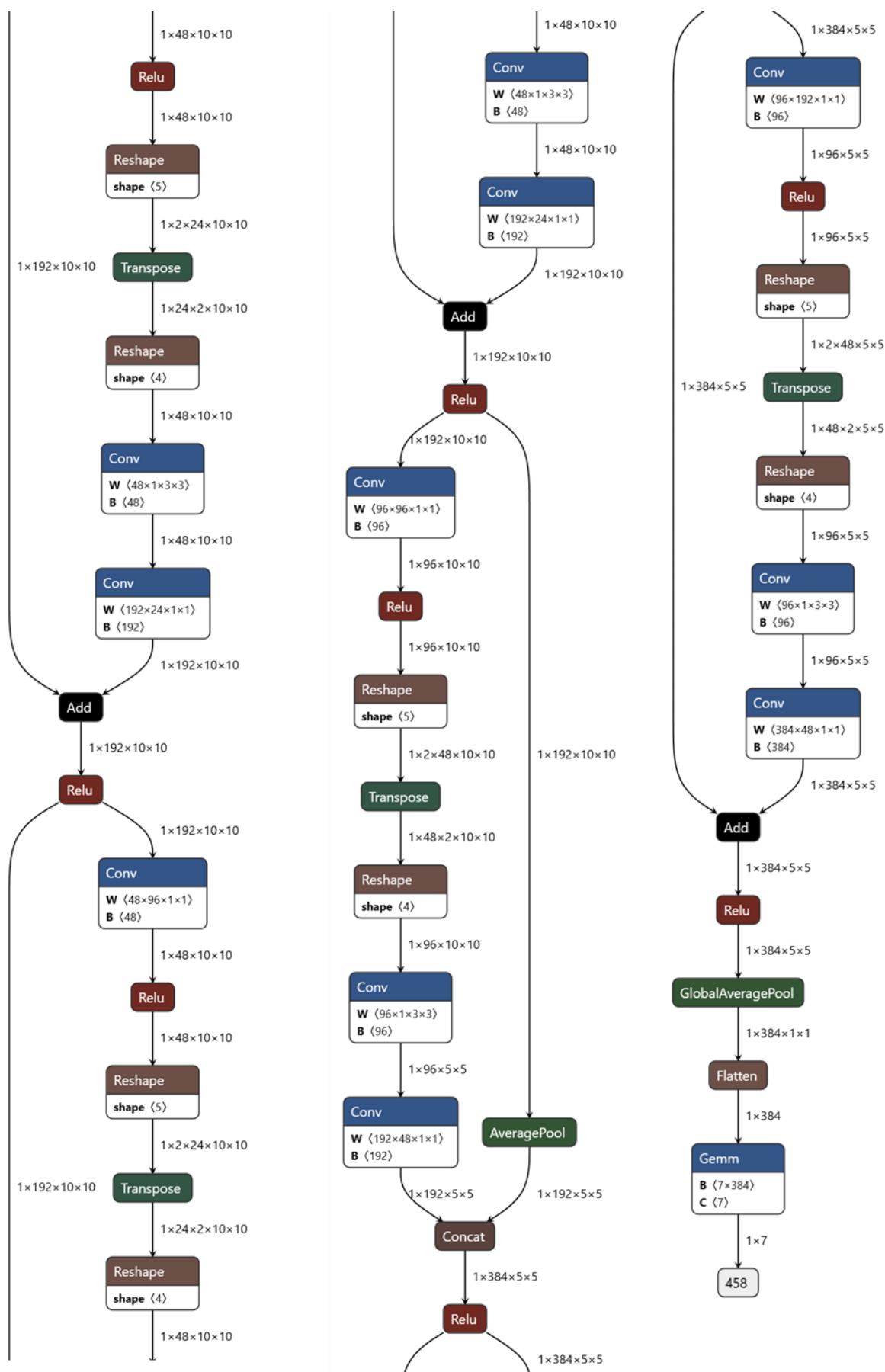


Рисунок В.6 – Архітектура «ShuffleNet V1*» (продовження)

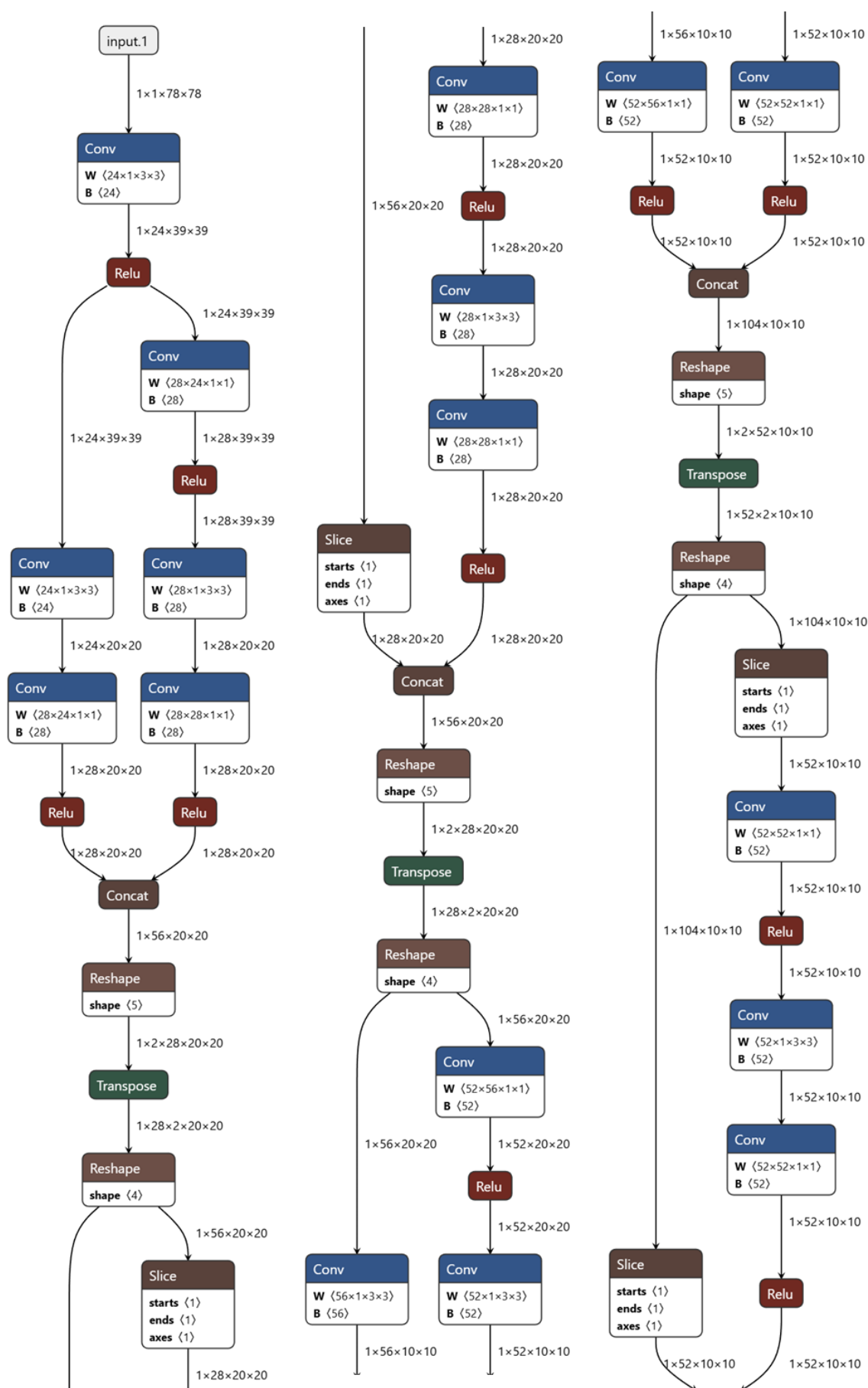


Рисунок В.7 – Архітектура «ShuffleNet V2*»

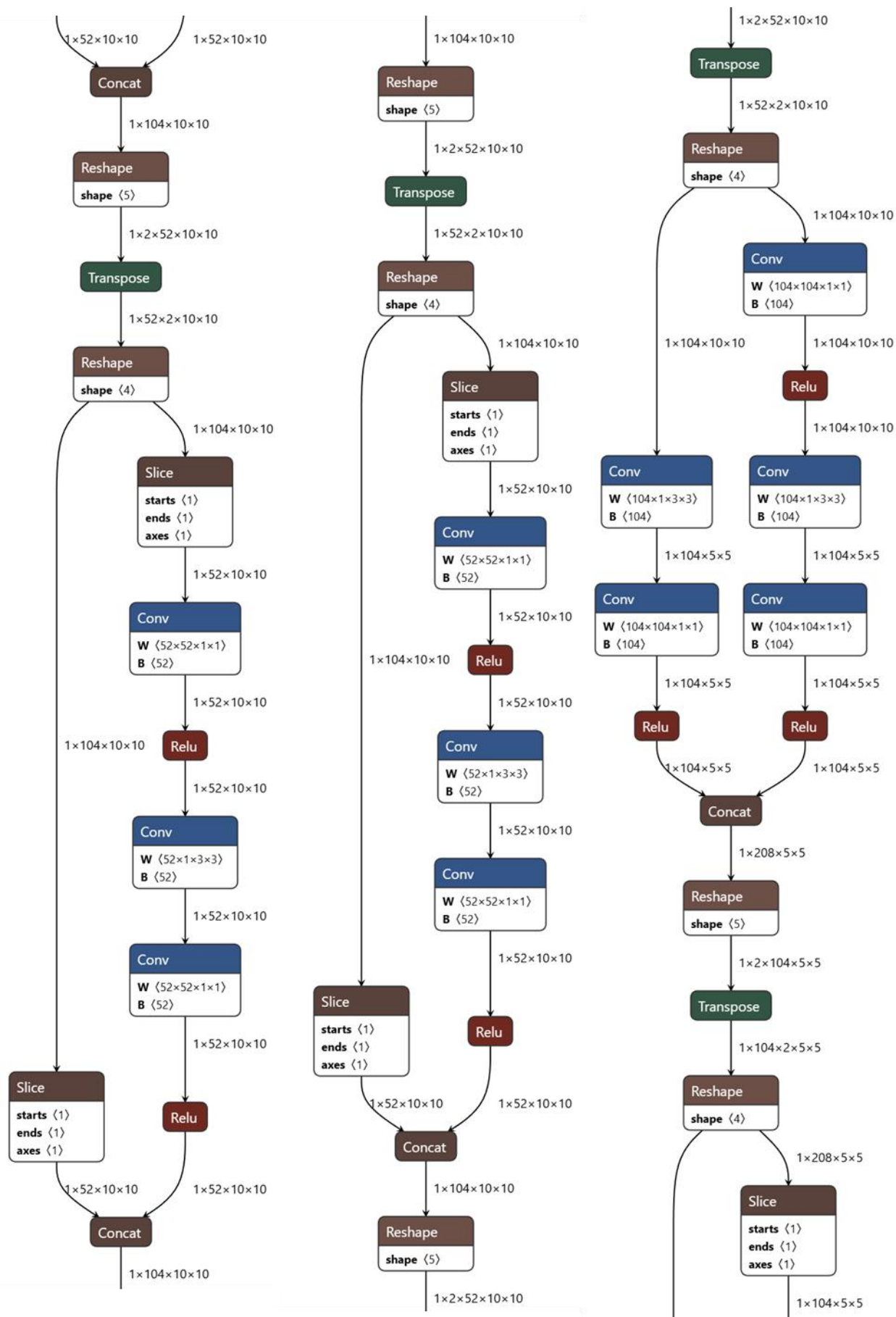


Рисунок В.7 – Архітектура «ShuffleNet V2*» (продовження)

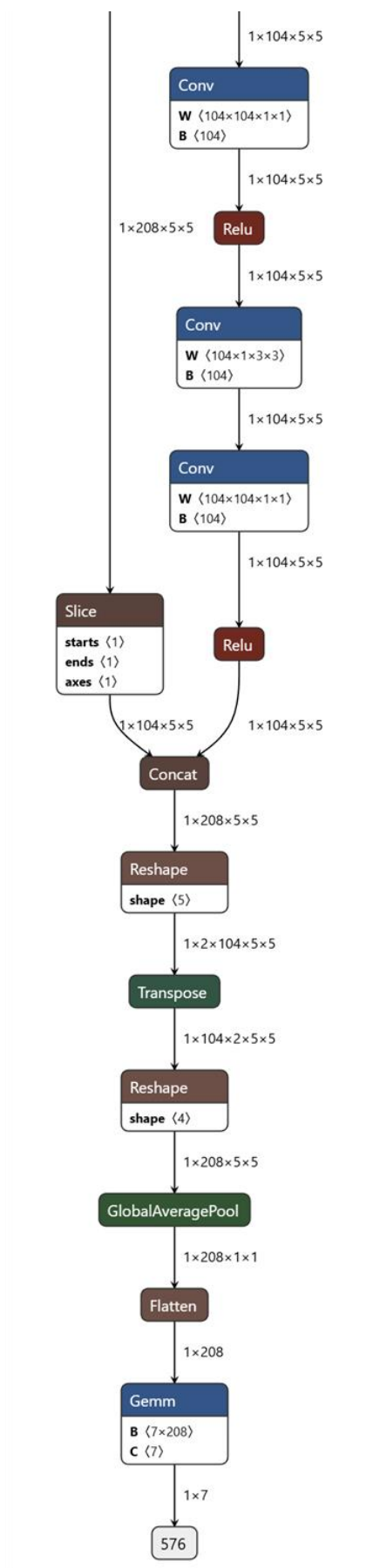


Рисунок В.7 – Архітектура «ShuffleNet V2*» (продовження)

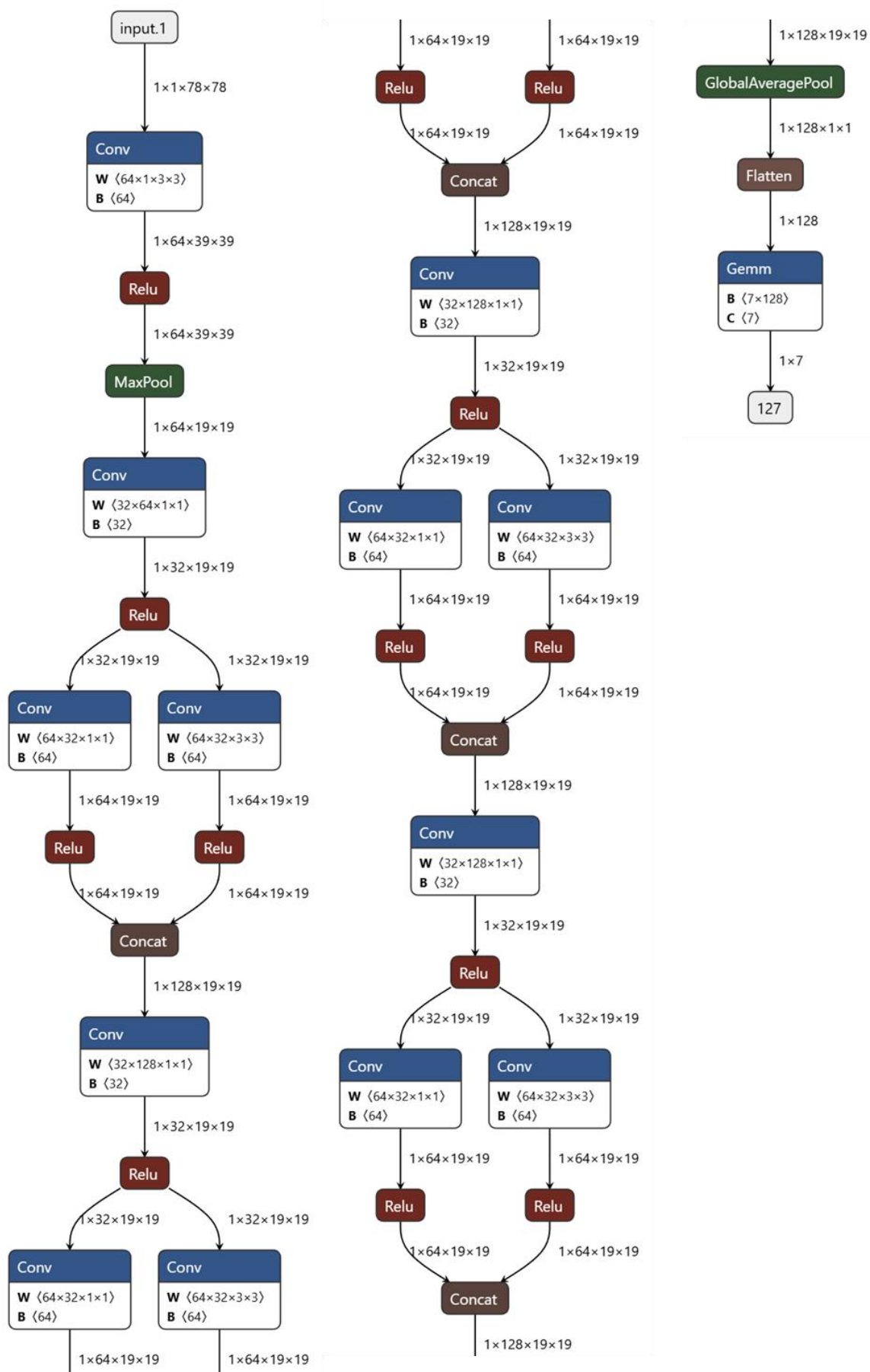


Рисунок В.8 – Архітектура «SqueezeNet*»

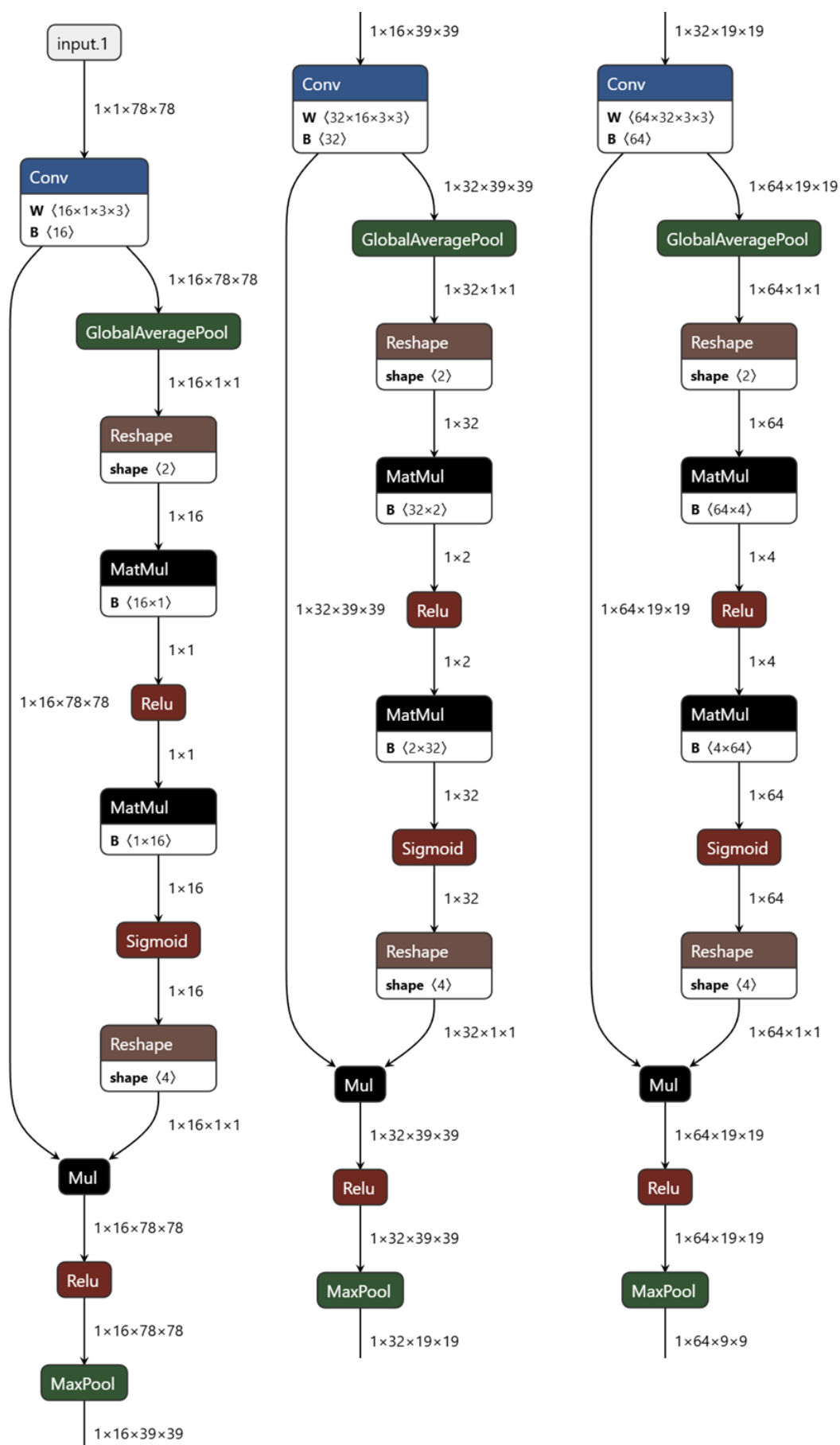


Рисунок В.9 – Архітектура «SE-Net*»

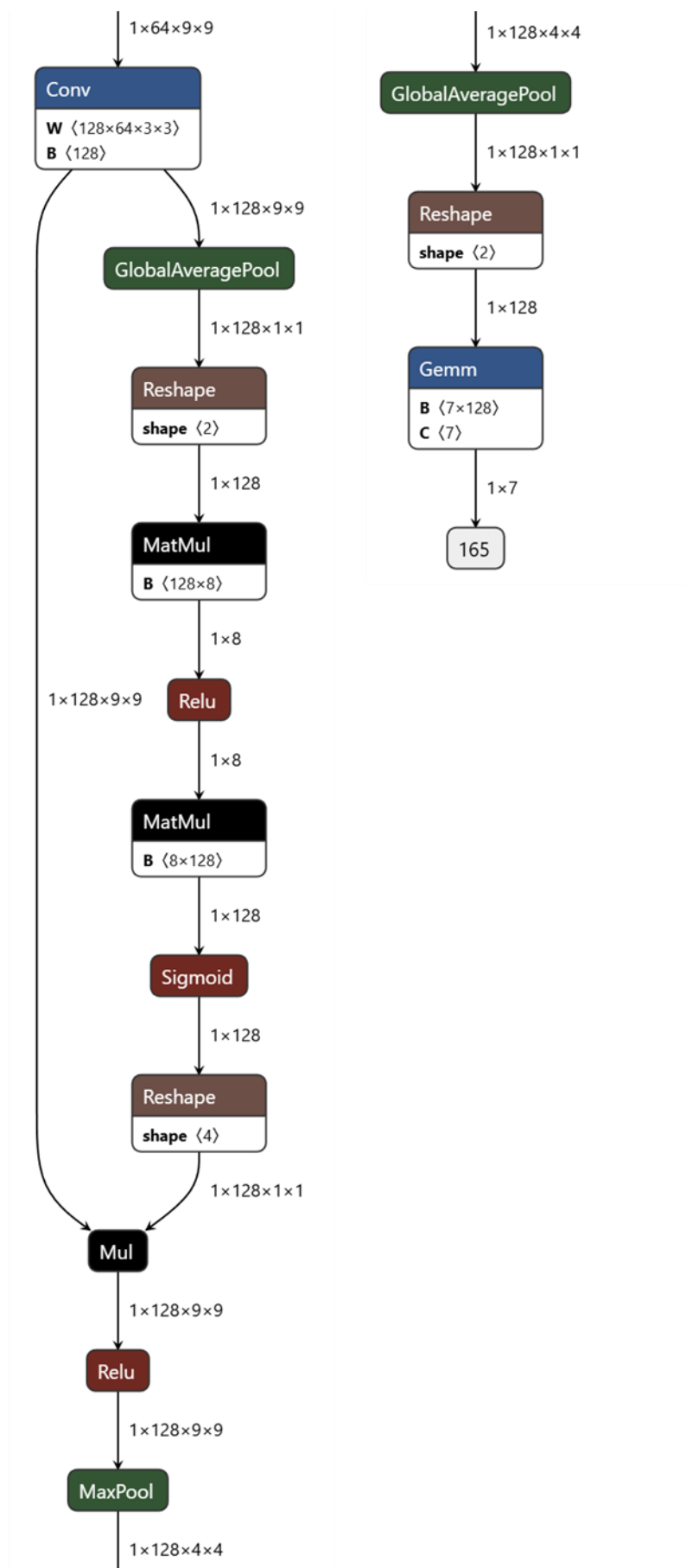
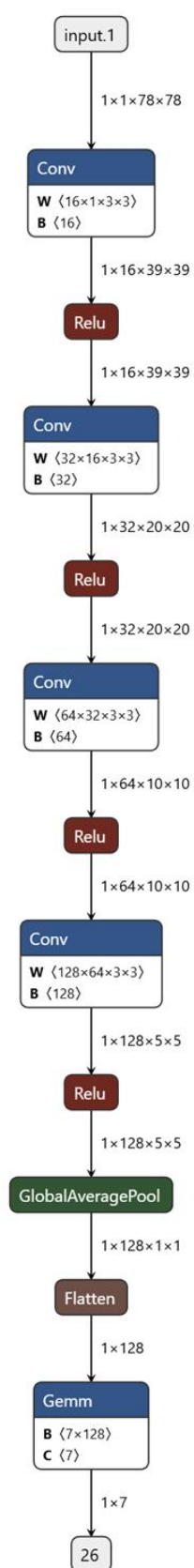
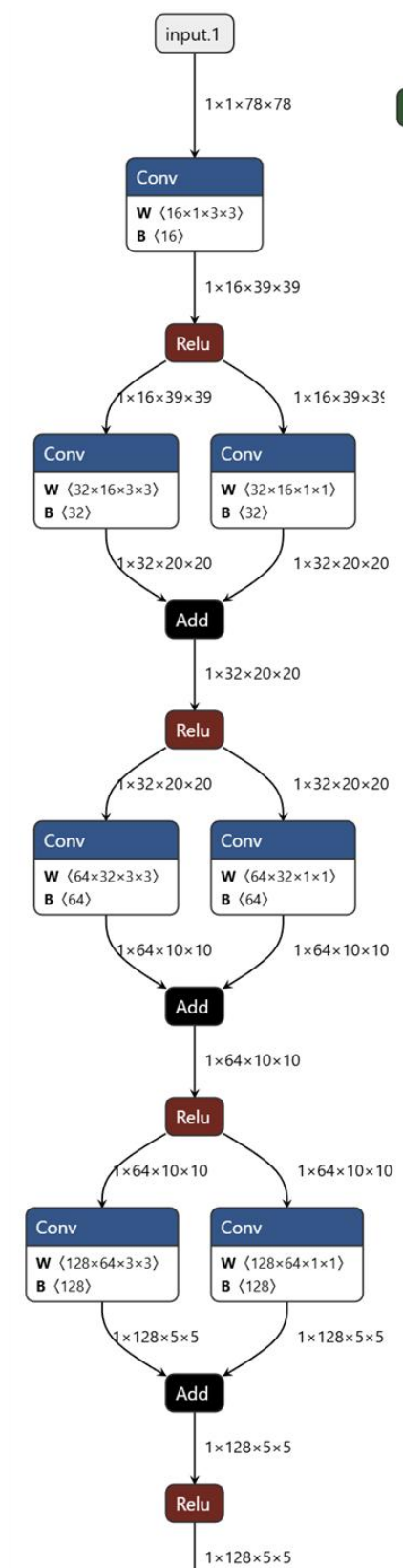


Рисунок В.9 – Архітектура «SE-Net*» (продовження)



а)



б)

а) під час інференсу (репараметризований варіант), б) під час навчання

Рисунок В.10 – Архітектура «RepVGG*»

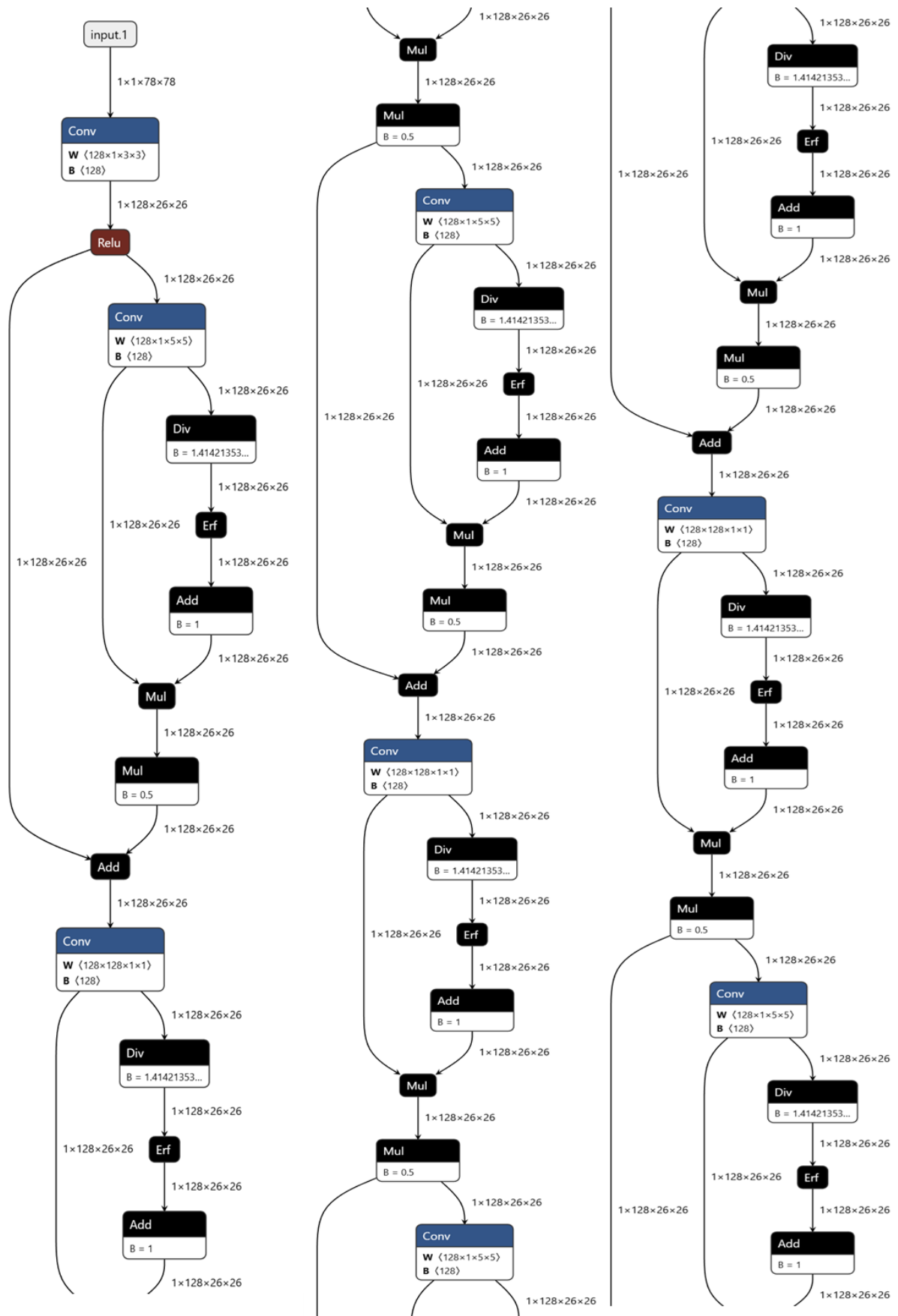


Рисунок В.11 – Архітектура «ConvMixer*»

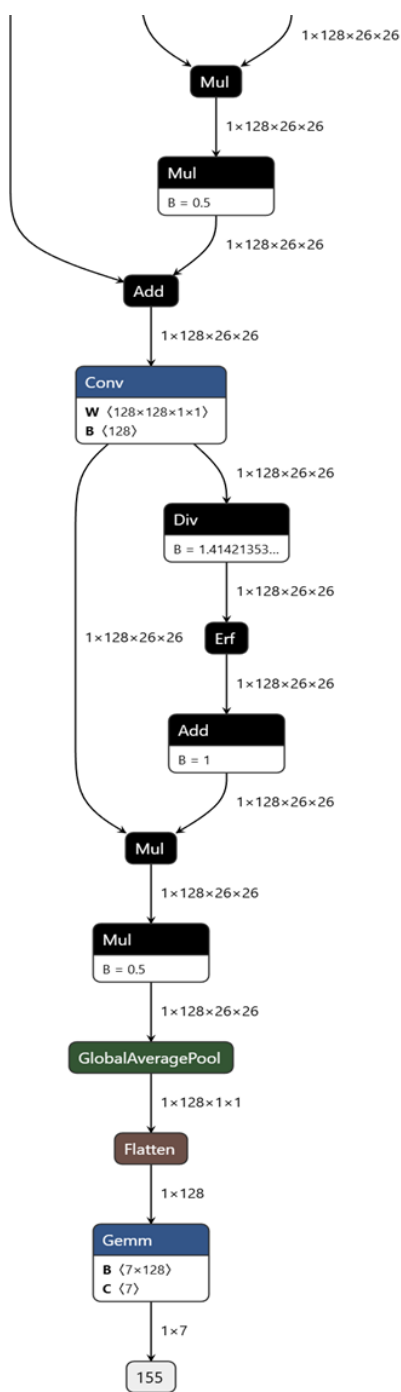


Рисунок В.11 – Архітектура «ConvMixer*» (продовження)

ДОДАТОК Г

Графіки скалярних метрик протягом навчання та валідації

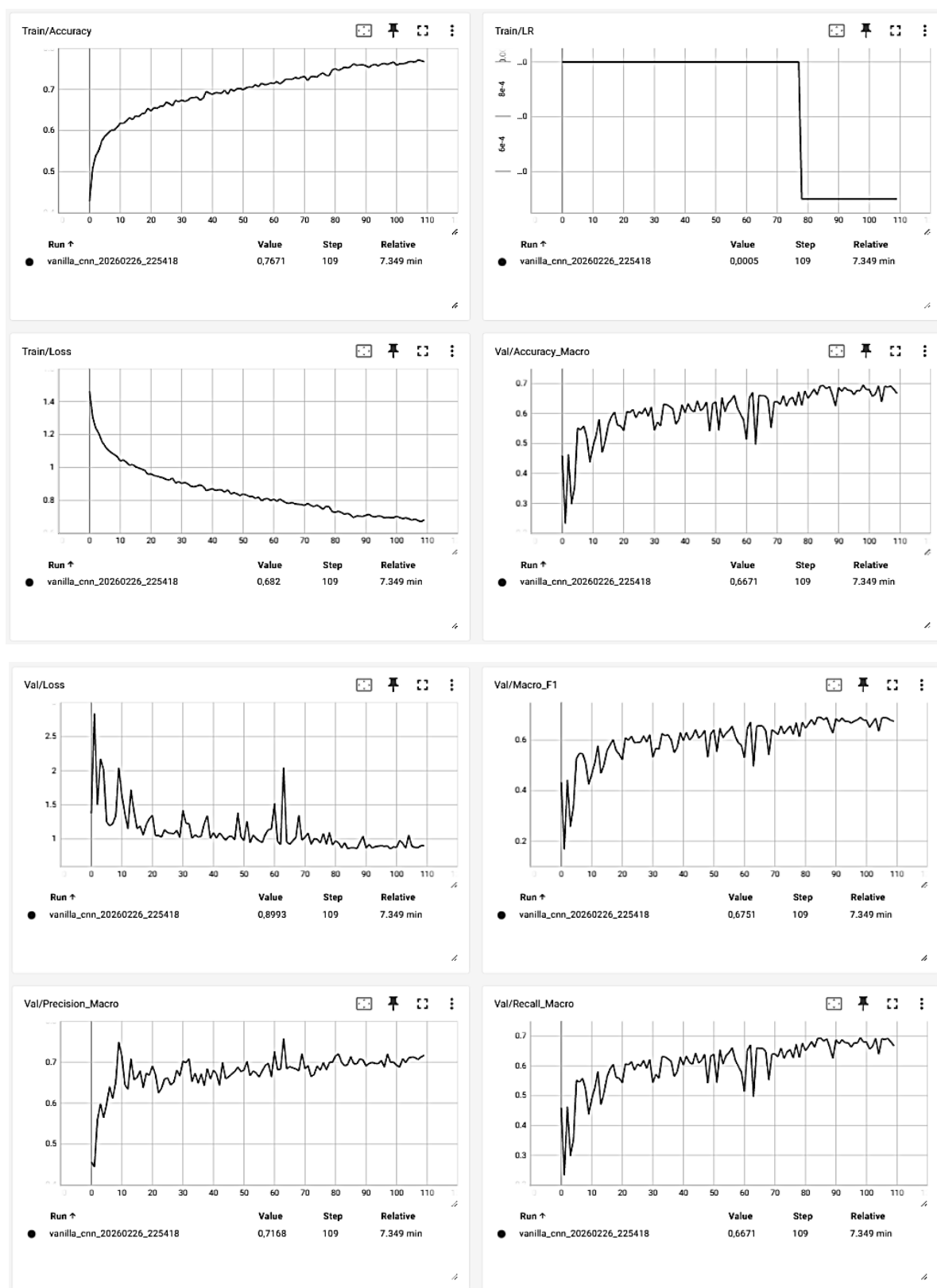


Рисунок Г.1 – Метрики моделі «Vanilla CNN»

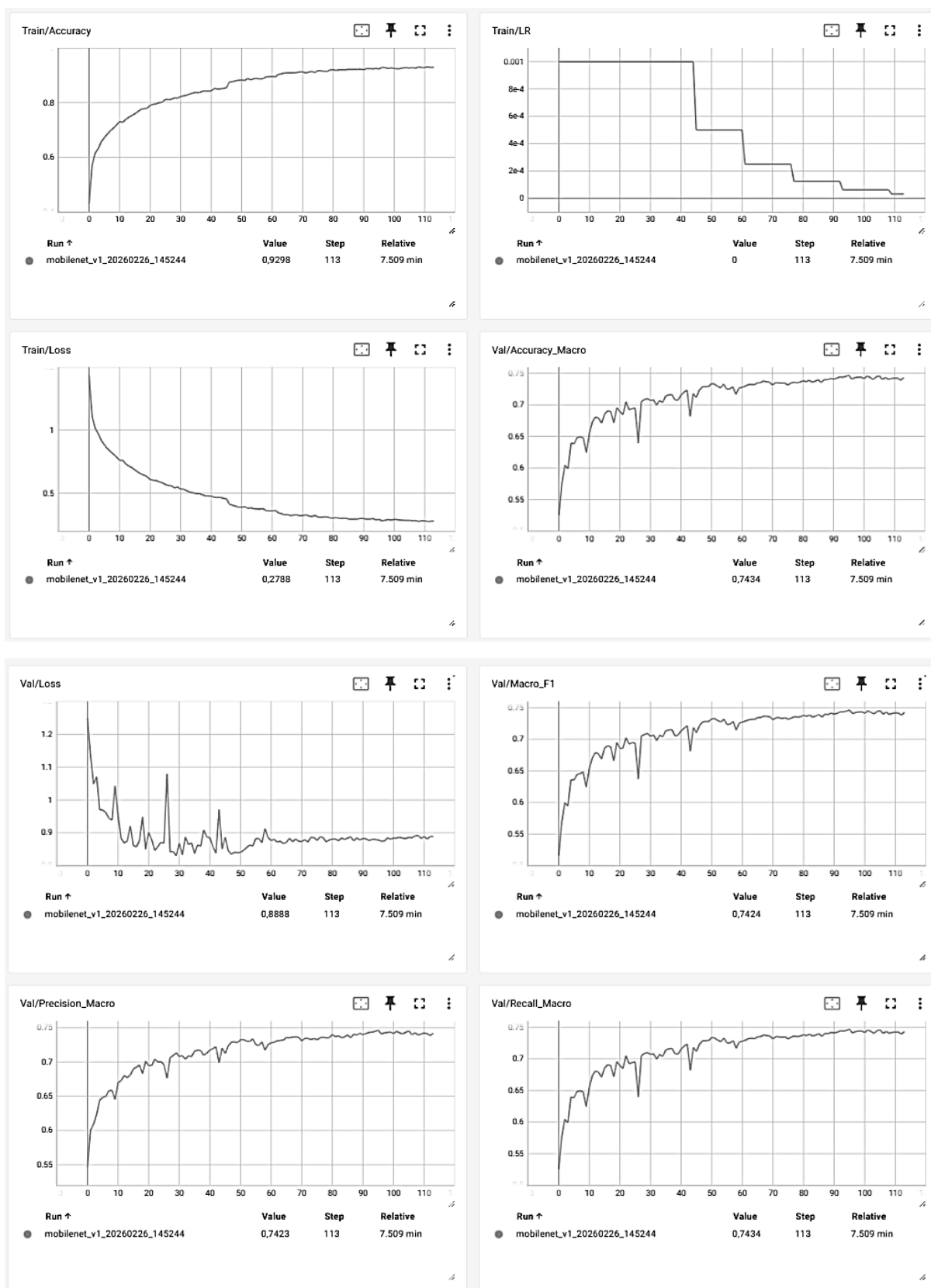


Рисунок Г.2 – Метрики моделі «Mobilenet V1*»

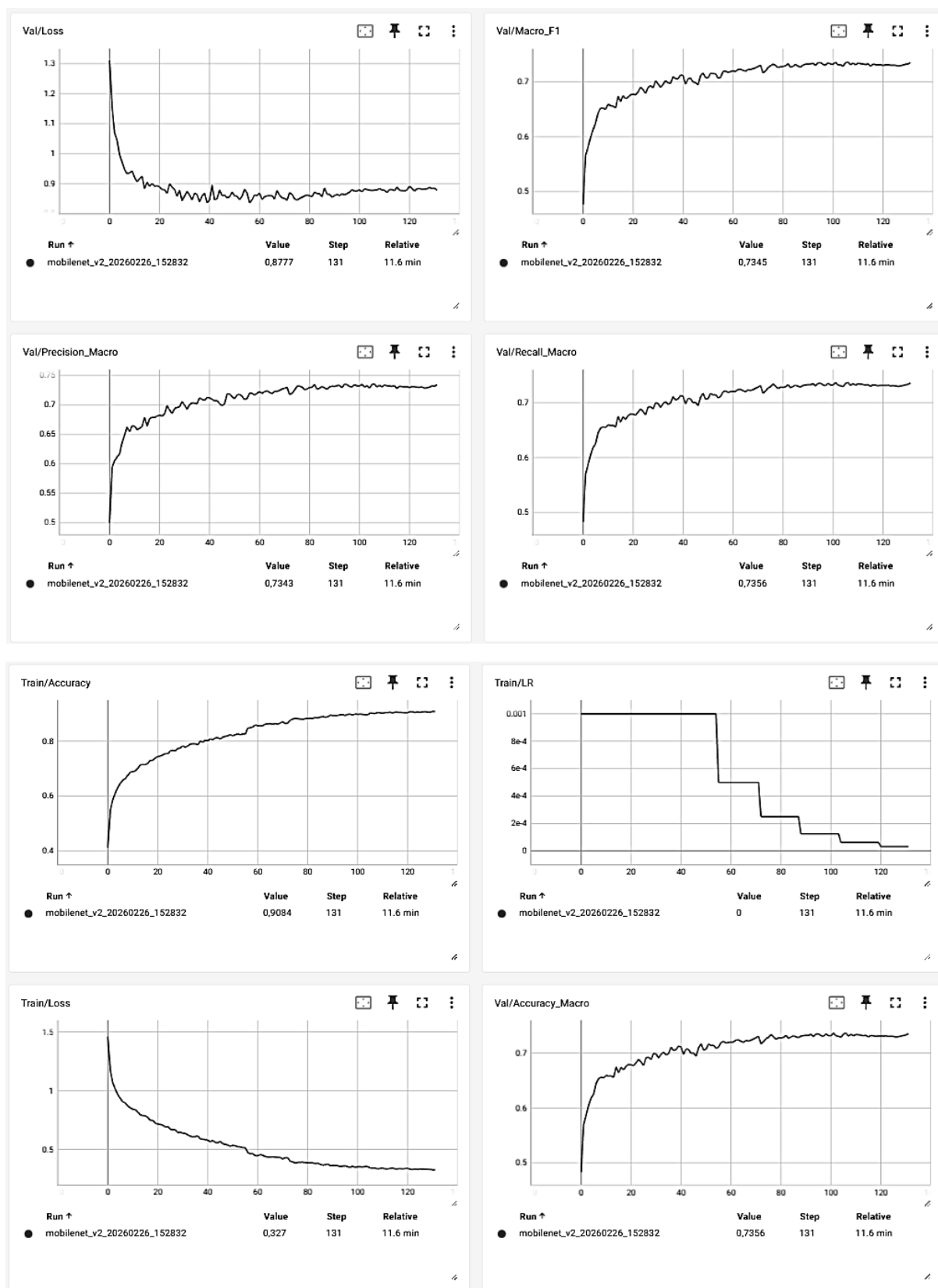


Рисунок Г.3 – Метрики моделі «Mobilenet V2*»

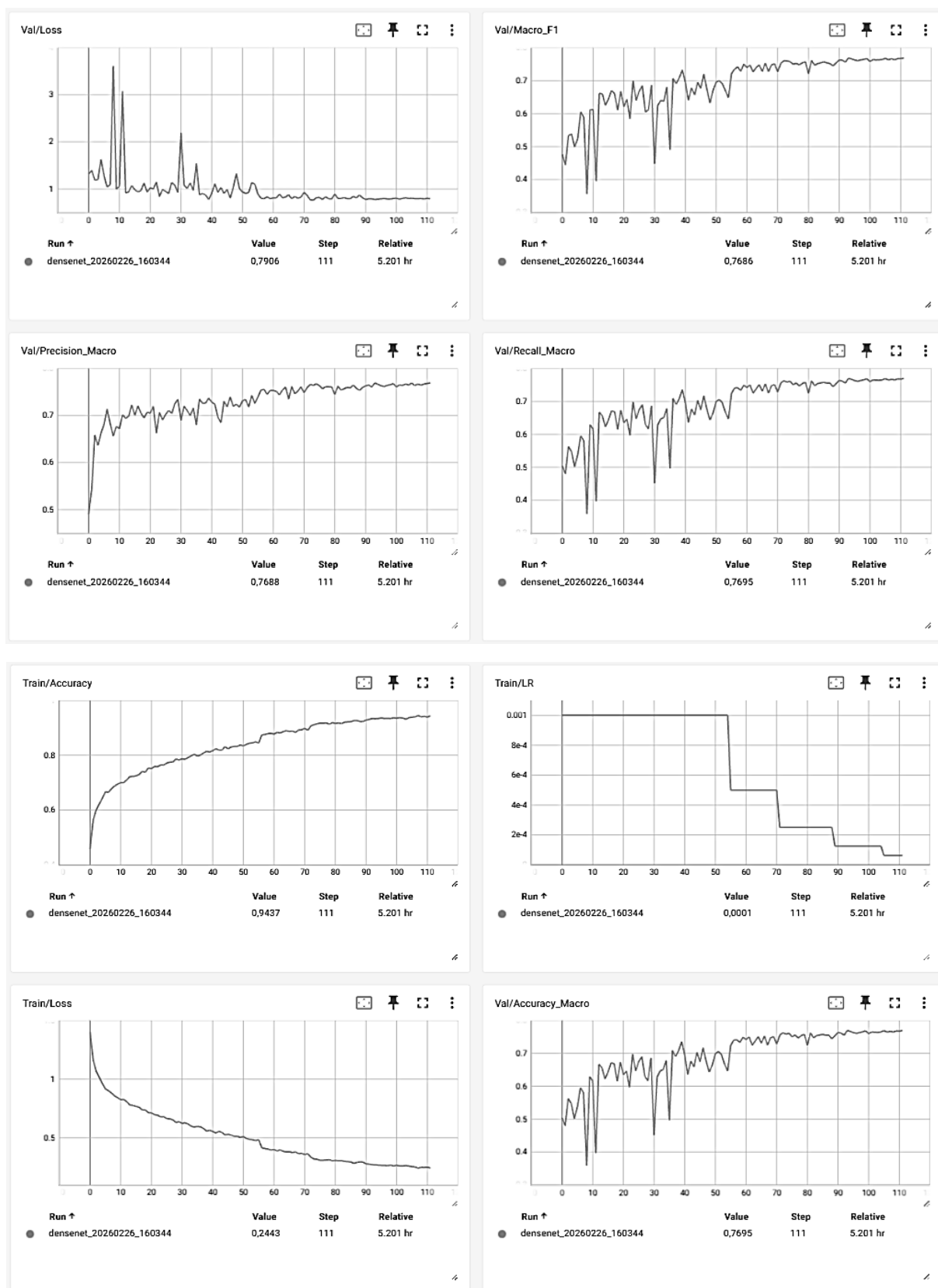


Рисунок Г.4 – Метрики моделі «DenseNet*»

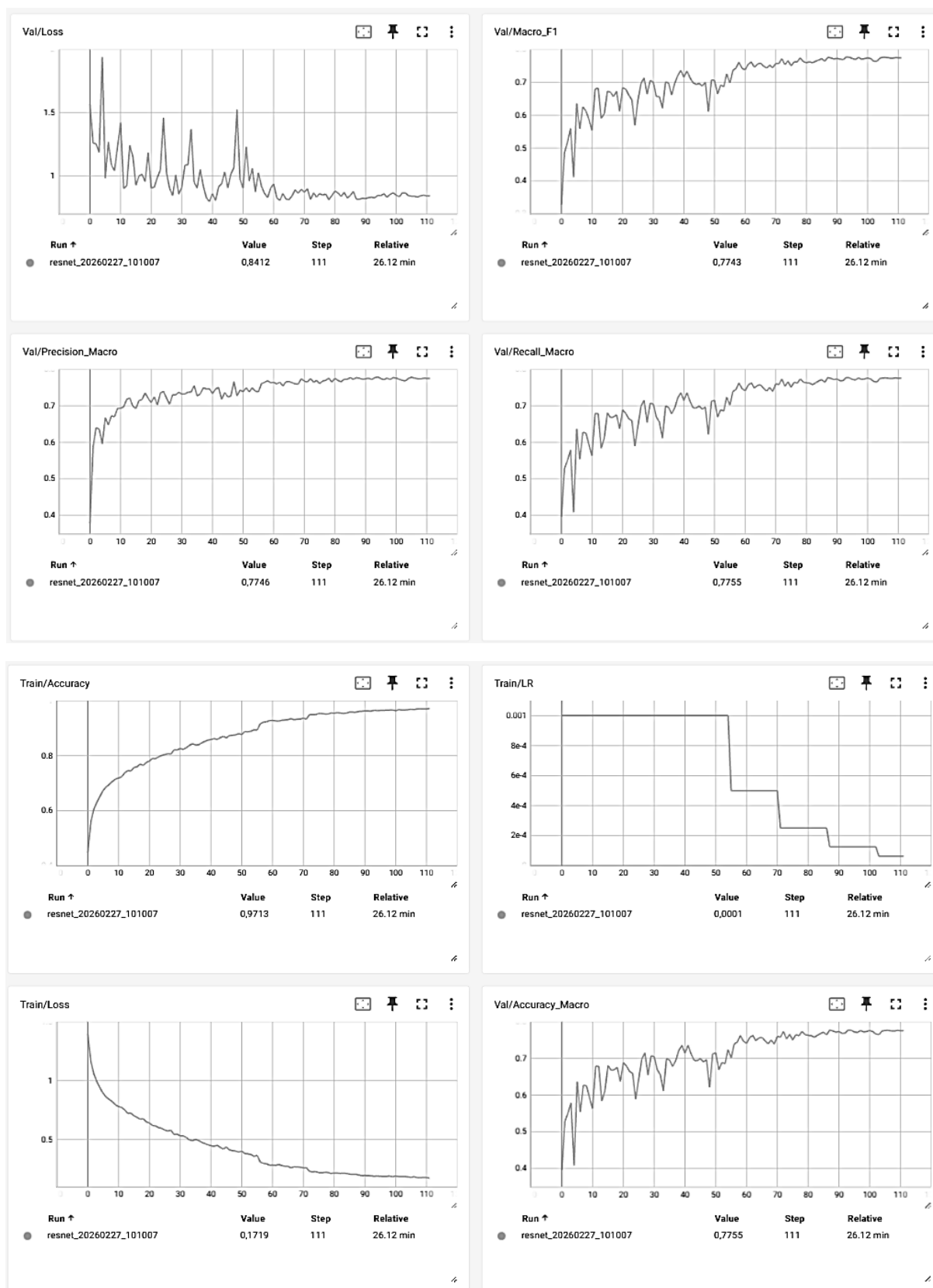


Рисунок Г.5 – Метрики моделі «ResNet*»

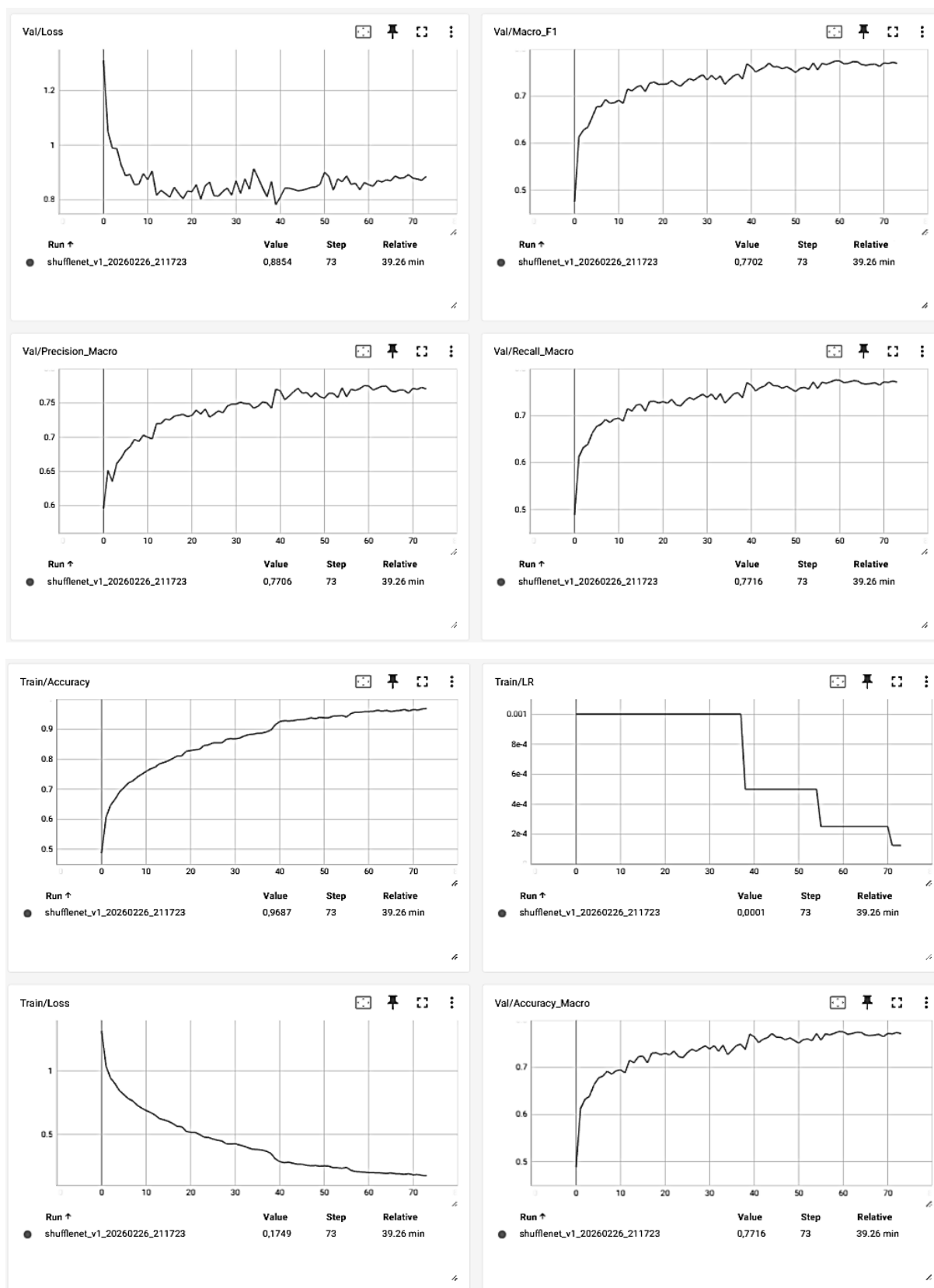


Рисунок Г.6 – Метрики моделі «ShuffleNet V1*»

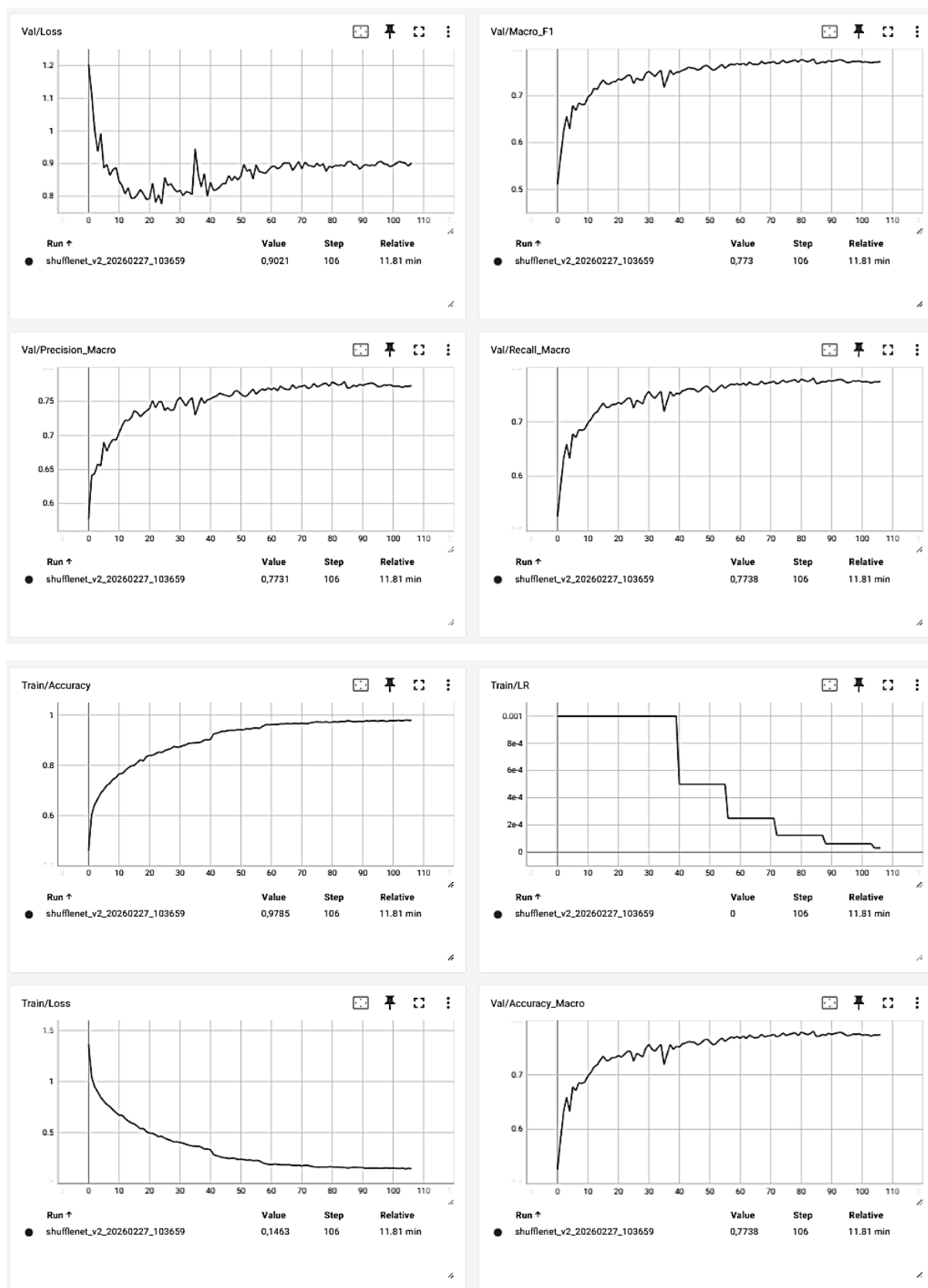


Рисунок Г.7 – Метрики моделі «ShuffleNet V2*»

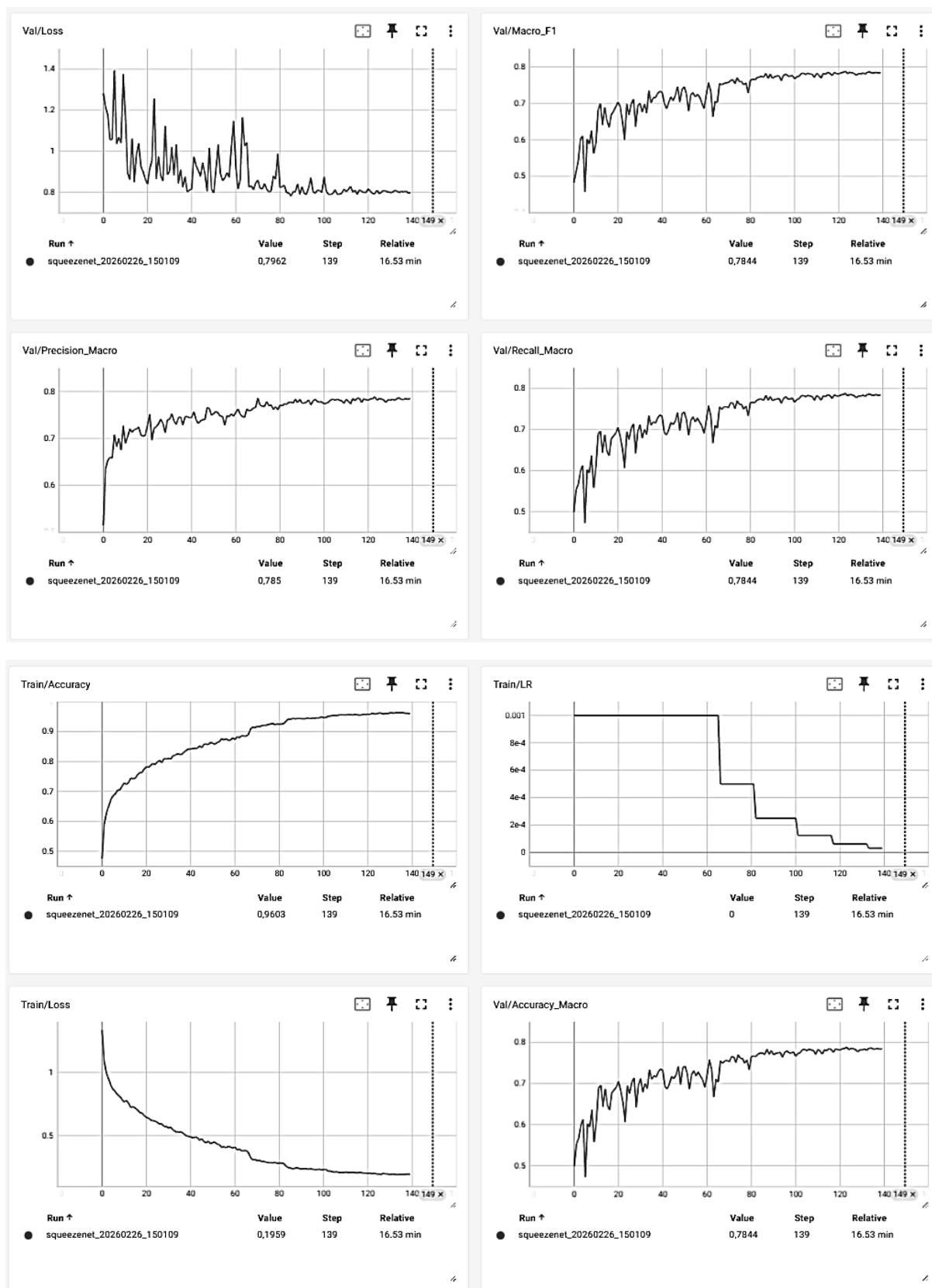


Рисунок Г.8 – Метрики моделі «SqueezeNet*»

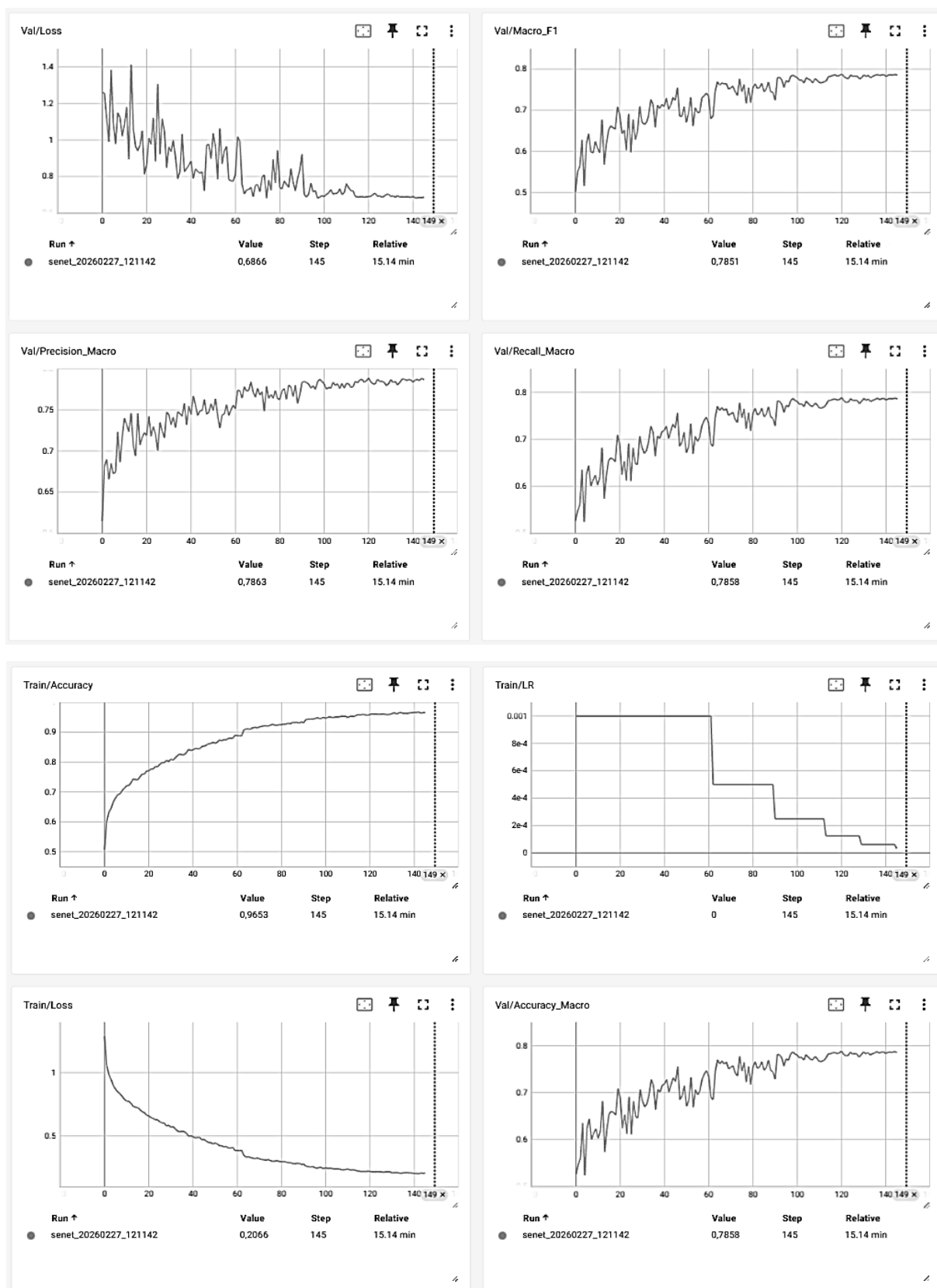


Рисунок Г.9 – Метрики моделі «SE-Net*»

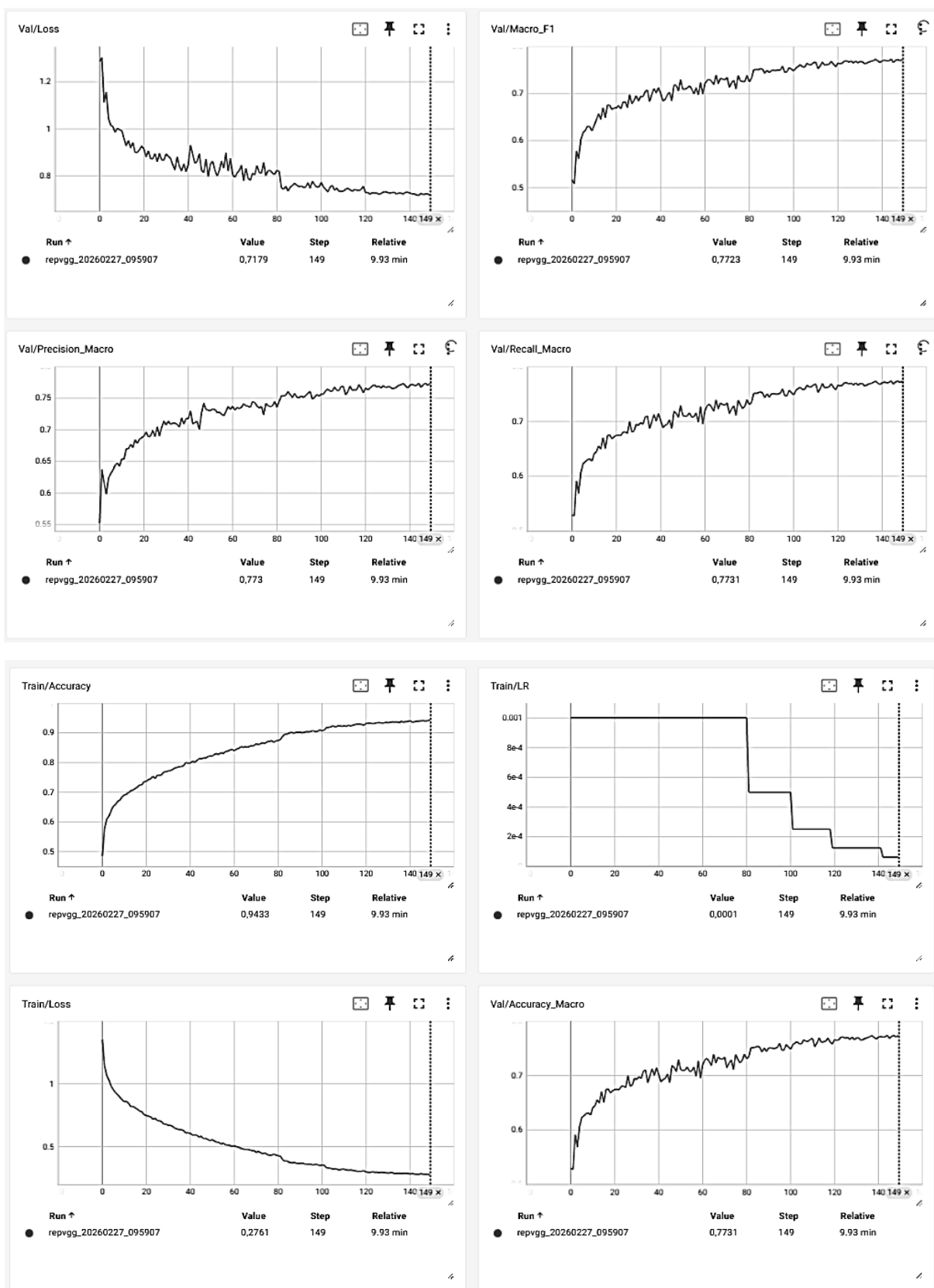


Рисунок Г.10 – Метрики моделі «RepVGG*»

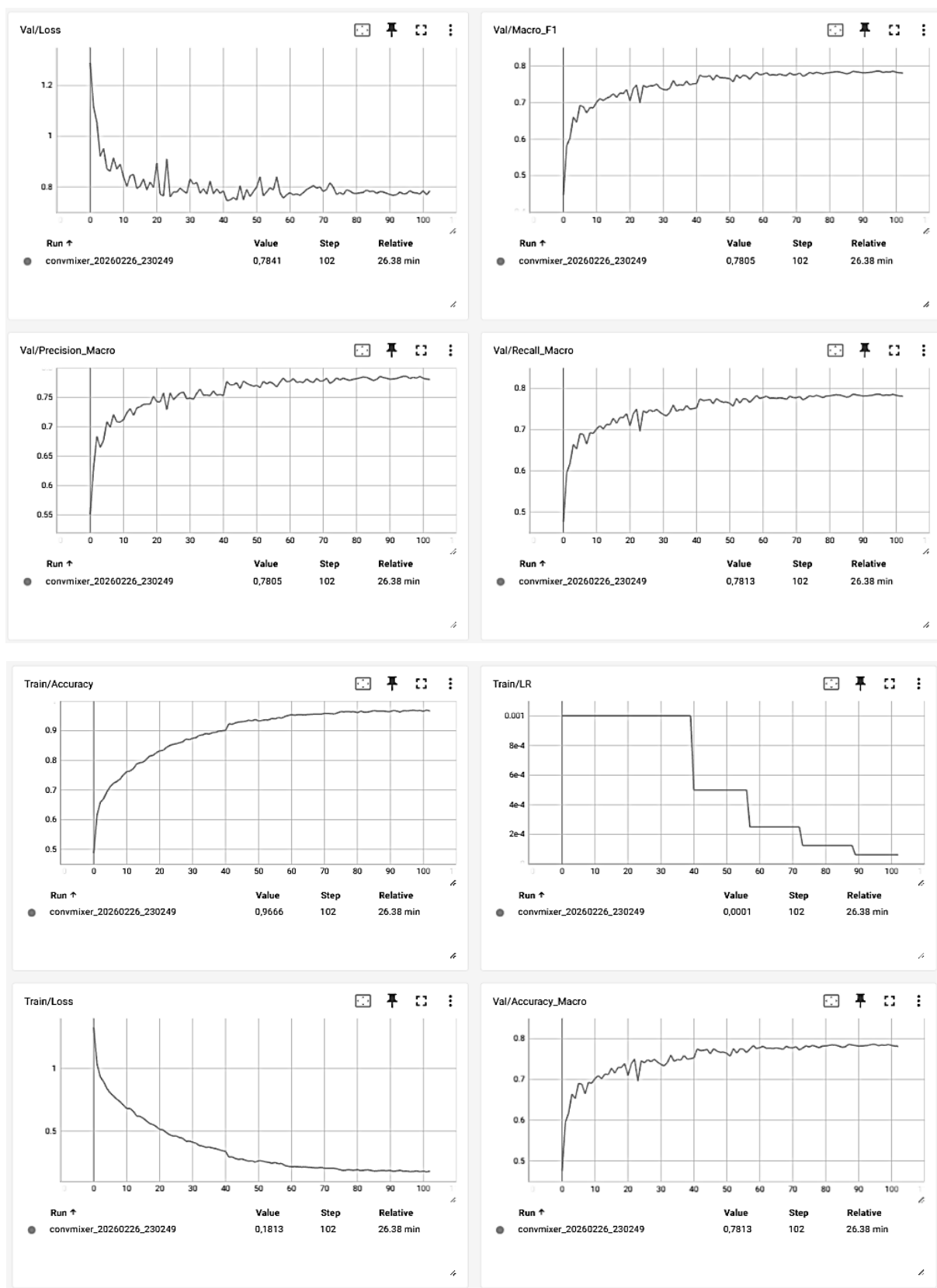


Рисунок Г.11 – Метрики моделі «ConvMixer*»

ДОДАТОК Д

Візуалізація матриць помилок, отриманих при тестуванні моделей

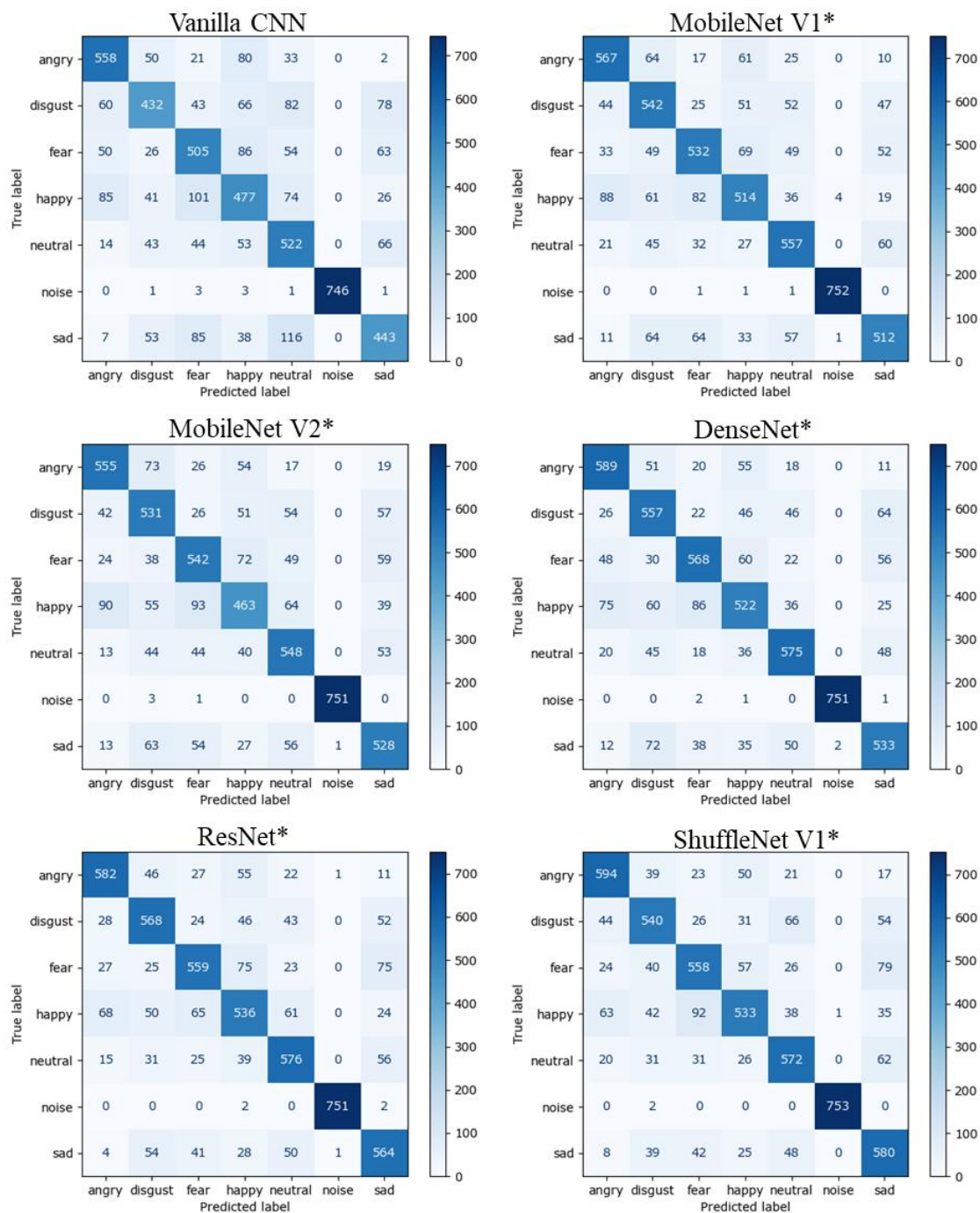


Рисунок Д.1 – Матриці помилок

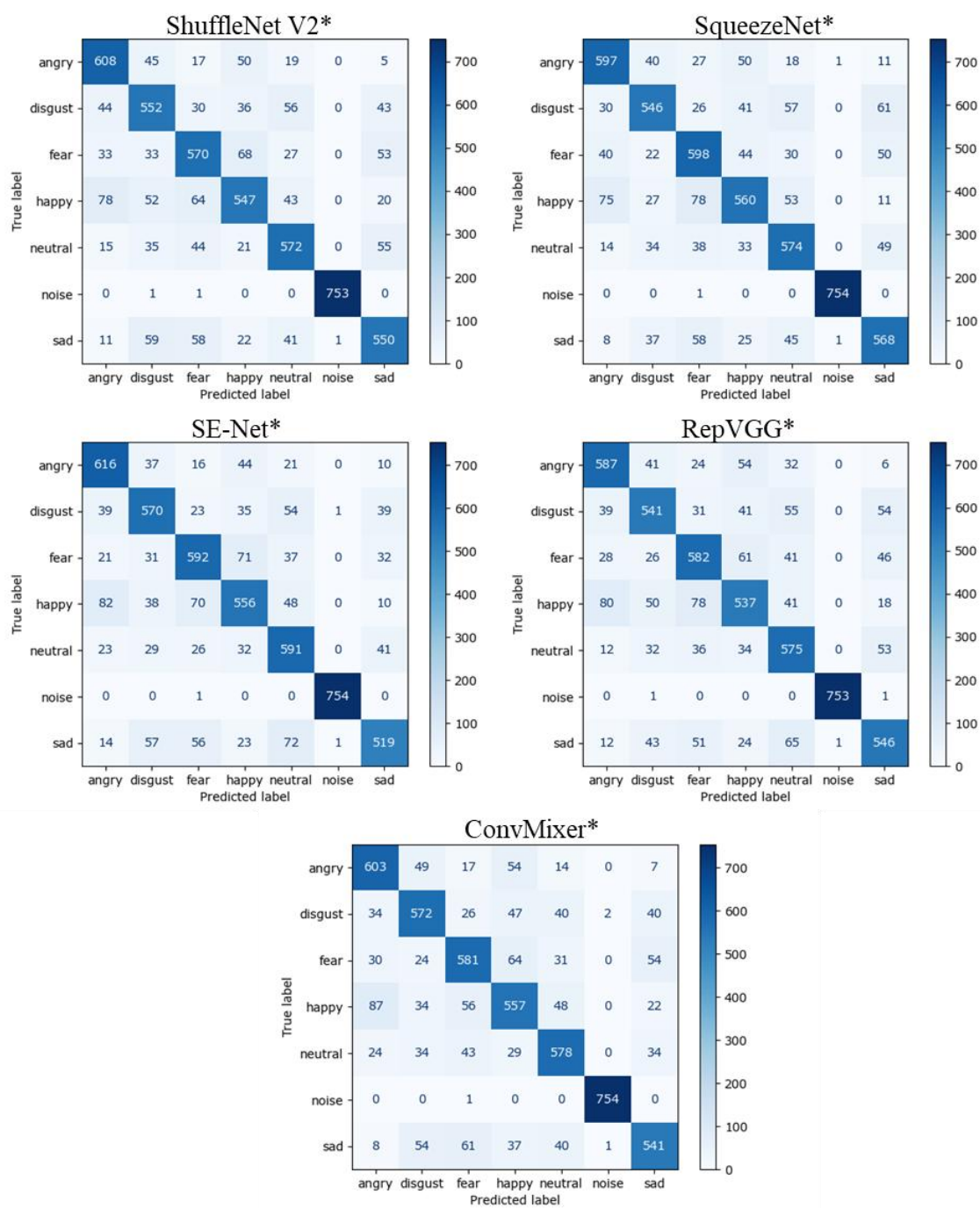


Рисунок Д.1 – Матриці помилок (продовження)

ДОДАТОК Е

Відомість кваліфікаційної роботи

Позначення					Найменування		Дод. відомості	
					Текстові документи			
1.					Пояснювальна записка		206 с.	