

Министерство науки и высшего образования РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«Владимирский государственный университет имени Александра
Григорьевича и Николая Григорьевича Столетовых»**
(ВлГУ)

ОТЧЕТ ПО КУРСОВОЙ РАБОТЕ
**«Разработка нейросетевой системы распознавания рукописного
кириллического текста на основе трансформера и сверточных нейронных
сетей»**
Специальность: 10.03.01 – «Информационная безопасность»

Руководитель

ст. пр. каф. ИЗИ

(должность)

(подпись, дата)

Мазурок Д.В.

(ФИО)

Исполнитель

ст. гр. ИБ-120

(должность)

(подпись, дата)

Рунов А.В.

(ФИО)

РЕФЕРАТ

Отчет 25 с., 26 рис., 12 источн.

РАСПОЗНАВАНИЕ РУКОПИСНОГО ТЕКСТА, КИРИЛЛИЧЕСКИЙ ТЕКСТ, НЕЙРОННАЯ СЕТЬ, ТРАНСФОРМЕР, СВЕРТОЧНАЯ НЕЙРОННАЯ СЕТЬ, ГИБРИДНАЯ АРХИТЕКТУРА, ОБРАБОТКА ИЗОБРАЖЕНИЙ, ПРЕДОБРАБОТКА ДАННЫХ, АУГМЕНТАЦИЯ ДАННЫХ, МЕТРИКИ КАЧЕСТВА, ИЗВЛЕЧЕНИЕ ПРИЗНАКОВ

Целью данной работы является проектирование и реализация гибридной нейронной сети, сочетающей трансформеры и свёрточные слои, для эффективного распознавания рукописного кириллического текста.

По ходу выполнения работы решались следующие **задачи**:

- 1) Провести сбор и подготовку набора данных с образцами рукописного кириллического текста. Выполнить предобработку данных для повышения качества обучения модели;
- 2) Разработать гибридную архитектуру нейронной сети, объединяющую трансформерные и свёрточные слои;
- 3) Осуществить обучение разработанной нейронной сети на подготовленном наборе данных. Провести тестирование модели и оценить ее производительность по метрикам точности распознавания.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	4
1 Сбор, подготовка, предобработка данных	5
1.1 Исходный набор данных	5
1.2 Исследование набора данных	5
1.3 Предобработка данных	7
2 Архитектура модели	14
3 Обучение и тестирование модели	20
3.1 Обучение	20
3.2 Тестирование	21
ЗАКЛЮЧЕНИЕ.....	24
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	25

ВВЕДЕНИЕ

Целью данной работы является проектирование и реализация гибридной нейронной сети, сочетающей трансформеры и свёрточные слои, для эффективного распознавания рукописного кириллического текста.

По ходу выполнения работы решались следующие **задачи**:

- 1) Провести сбор и подготовку набора данных с образцами рукописного кириллического текста. Выполнить предобработку данных для повышения качества обучения модели;
- 2) Разработать гибридную архитектуру нейронной сети, объединяющую трансформерные и сверточные слои;
- 3) Осуществить обучение разработанной нейронной сети на подготовленном наборе данных. Провести тестирование модели и оценить ее производительность по метрикам точности распознавания.

1 Сбор, подготовка, предобработка данных

1.1 Исходный набор данных

В данной работе использовался набор данных, содержащий образцы рукописного кириллического текста для задач распознавания текста (OCR). Этот набор состоит из 73830 сегментов рукописного текста на русском языке и разделен на обучающую и тестовую выборки в соотношении 97% и 3% соответственно.

Каждый пример данных представляет собой изображение текстового фрагмента на русском языке, состоящего не более чем из 40 символов, написанных от руки.

Общий размер набора данных достигает приблизительно 2 ГБ.

Из исходной обучающей выборки, составляющей было выделено еще 3% данных для использования в качестве валидационной выборки.

Кроме того, для удобства работы с данными, я произвел конвертацию исходных файлов из формата TSV (табличные значения, разделенные табуляцией) в более распространенный формат CSV (значения, разделенные запятыми). При этом была добавлена строка с названиями столбцов для наглядности. Первый столбец содержит пути к изображениям, а второй - соответствующие текстовые метки.

Для обучения модели была использована лишь часть исходной обучающей выборки - 10% от ее полного объема (7068 примеров). Это было сделано в связи с ограниченными вычислительными ресурсами, доступными для проведения обучения.

1.2 Исследование набора данных

Рассмотрим основные характеристики использованных данных для всех трех выборок (обучающей, валидационной и тестовой):

- 1) Общий размер набора данных составляет 10204 примера.
- 2) Максимальная длина текстовой метки (рукописного текстового фрагмента) равна 38 символам.
- 3) Наиболее часто встречающимся символом в наборе данных является буква "о", которая присутствует 7957 раз.
- 4) Наименее часто встречающимся символом является символ ">", который встречается лишь один раз во всем наборе.
- 5) Самой распространенной текстовой меткой является слово "что", которое встречается 44 раза.
- 6) Наименее распространенной текстовой меткой является "Васильевича", которая встречается только один раз.

На рис. 1 представлено распределения частот символов в наборе.

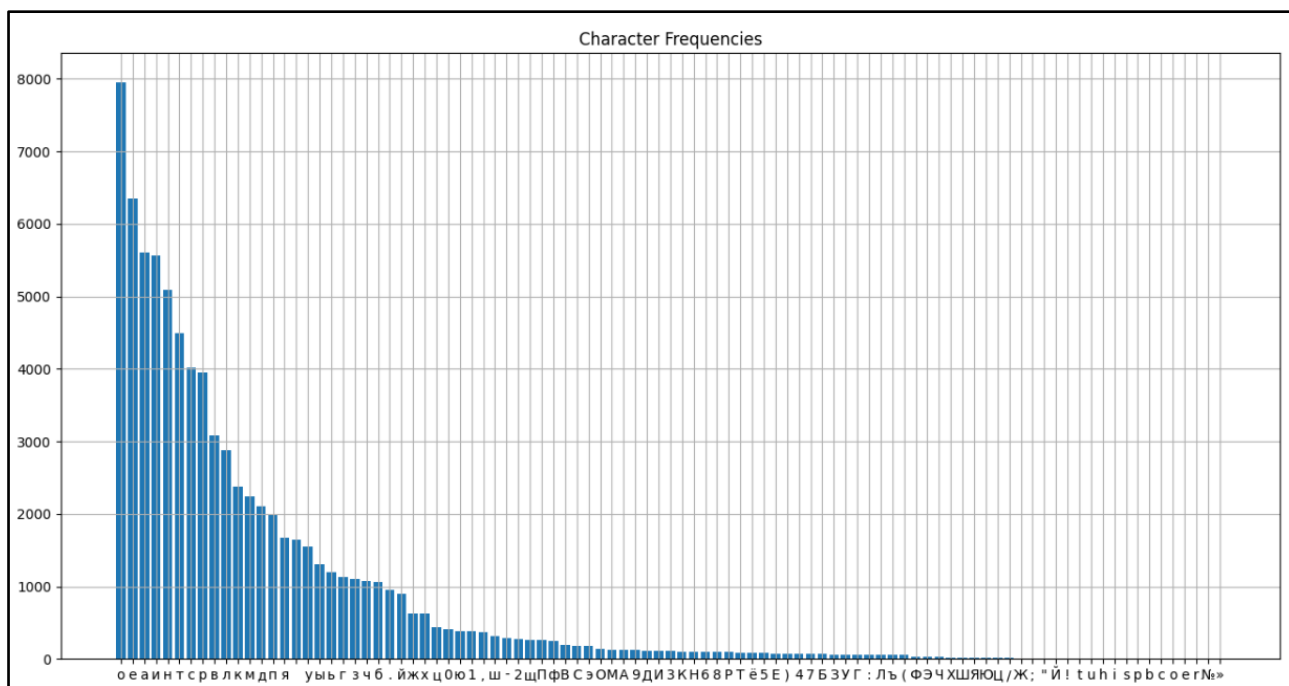


Рисунок 1 – Частотное распределение символов в наборе

На рис. 2 представлено распределение длин текстовых меток (в символах):

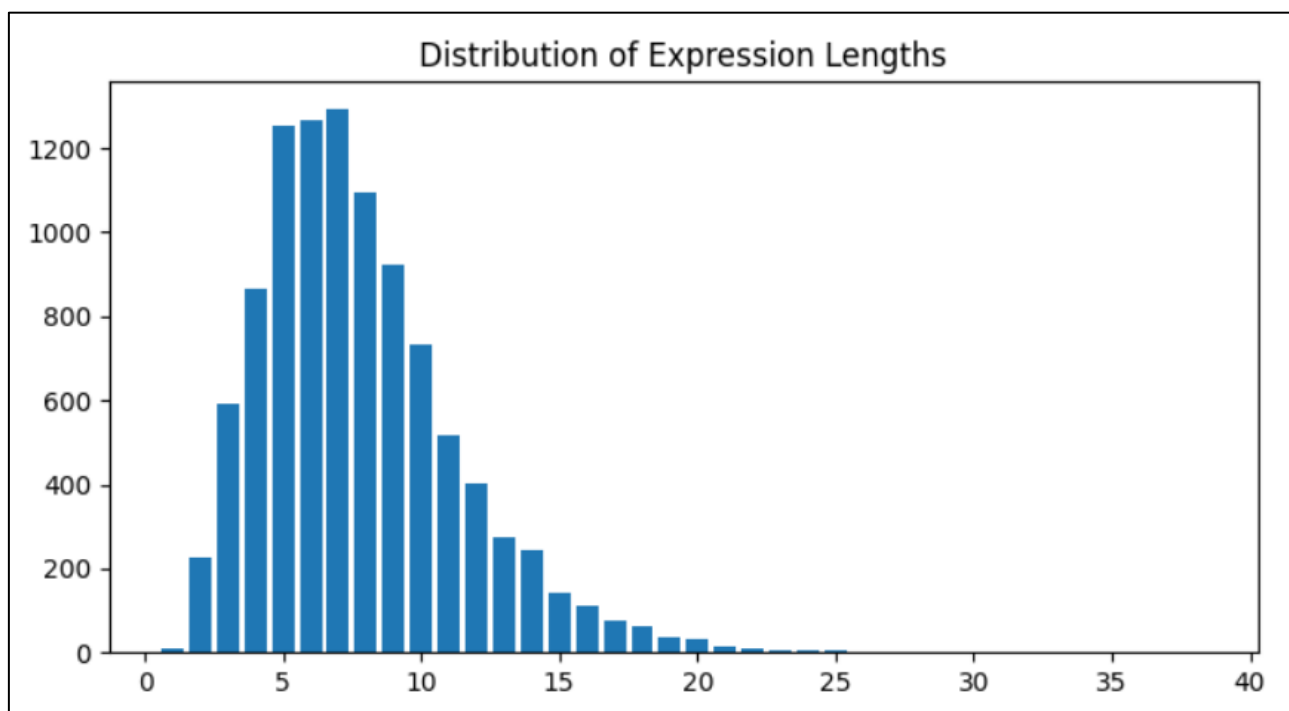


Рисунок 2 – Распределение длины текстовых меток в символах; наиболее частая длина – от 5 до 10 символов

На рис. 3 представлены примеры изображений и их метки.

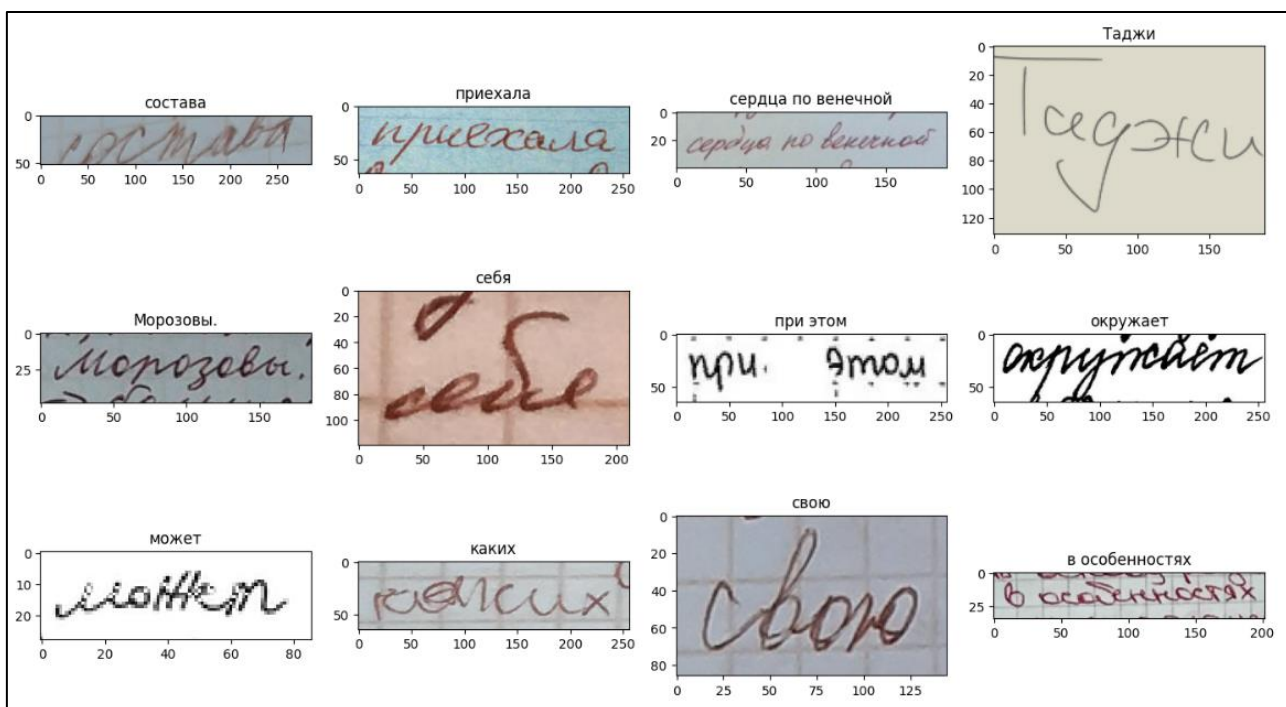


Рисунок 3 – Примеры изображений с соответствующими им метками

Изображения в наборе данных имеют следующие характеристики:

- 1) Минимальная высота изображения: 16 пикселей
- 2) Максимальная высота изображения: 827 пикселей
- 3) Минимальная ширина изображения: 32 пикселя
- 4) Максимальная ширина изображения: 1290 пикселей
- 5) Изображения являются цветными и содержат 3 цветовых канала.

1.3 Предобработка данных

На первом этапе из обучающей выборки были удалены строки, содержащие символы, не входящие в принятый для данной задачи алфавит (рис. 4). Это позволило очистить данные от нежелательных символов и сосредоточиться на релевантных образцах рукописного текста.

```
[9]: # List of characters that are used for recognition
alphabet = [' ', '!', '"', '%', '(', ')', ',', '-', '+', '=', '.', '/', '"', '"',
            '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ':', ';', '?', '[',
            ']', '«', '»', 'А', 'Б', 'В', 'Г', 'Д', 'Е', 'Ж', 'З', 'И', 'Й', 'К',
            'Л', 'М', 'Н', 'О', 'П', 'Р', 'С', 'Т', 'У', 'Ф', 'Х', 'Ц', 'Ч', 'Ш',
            'Щ', 'Э', 'Ю', 'Я', 'а', 'б', 'в', 'г', 'д', 'е', 'ж', 'з', 'и', 'й',
            'к', 'л', 'м', 'н', 'о', 'п', 'р', 'с', 'т', 'у', 'ф', 'х', 'ц', 'ч',
            'ш', 'щ', 'ъ', 'ы', 'ь', 'э', 'ю', 'я', 'ё']
```

Рисунок 4 – Набор символов, используемый для распознавания

Далее к набору символов были добавлены служебные токены: '_EOS' (конец последовательности), 'PAD' (символ заполнения) и 'SOS' (начало последовательности) (рис. 5). Эти

специальные символы необходимы для корректной работы определенных архитектур нейронных сетей и правильного форматирования входных данных.

```
[38]: # Add special tokens
alphabet = ['_PAD', '_SOS'] + alphabet + ['_EOS']

# Expected {'_EOS', '_PAD', '_SOS'}
set(alphabet).difference(set(chars_train))
```

Рисунок 5 – Добавление спец символов в алфавит

Следующим шагом стала нормализация размеров изображений. Все изображения были приведены к единому размеру (64, 256, 3), где 64 и 256 – высота и ширина в пикселях соответственно, а 3 указывает на наличие трех цветовых каналов (рис. 6). Это обеспечило единообразие входных данных для обучения нейронной сети.

```
[41]: img_ = np.asarray(Image.open(img_names_train[0]).convert('RGB'))

[42]: print(f'Original shape of example image {img_.shape}')
      Original shape of example image (52, 286, 3)

[43]: Image.fromarray(img_)
[43]: 

[44]: processed_img_ = process_image(img_)

[45]: print(f'Reshaped shape of example image {processed_img_.shape}')
      Reshaped shape of example image (64, 256, 3)

[46]: Image.fromarray(processed_img_.astype(np.uint8))
[46]: 
```

Рисунок 6 – Пример изменения размера изображения

Кроме того, был реализован механизм сопоставления каждому уникальному символу индекса для удобства работы с данными. Были созданы функции для конвертации текстовых меток в списки соответствующих индексов (рис. 7, 8) и обратной операции – восстановления текстовых меток из списков индексов. Это упростило процесс подачи данных на вход нейронной сети и интерпретацию ее выходных данных.


```
[52]: def text_to_labels(text, char2idx):
      """
      Convert text to label vector based on char2idx mapping

      Args:
          text (str): Input text
          char2idx (dict): Mapping of characters to indexes

      Returns:
          labels (List[int]): List of character indexes
      """

      # Initialize labels vector
      labels = []

      # Add start-of-sentence token at start
      labels.append(char2idx['_SOS'])

      # Map characters to indexes
      for char in text:
          # Skip unknown characters
          if char in char2idx:
              labels.append(char2idx[char])

      # Add end-of-sentence token at end
      labels.append(char2idx['_EOS'])

      return labels
```

Рисунок 7 – Функция для перевода текстовых меток в список индексов по соответствующему словарию; в начало и конец последовательности добавляются спец символы

```
[65]: print(f'Example of text-to-label conversion "{train_dataset.text_labels[0]}" \
      -> {train_dataset[0]['label'].tolist()}')

      print(f'Example of label-to-text conversion {train_dataset[0]['label'].tolist()} \
      -> "{labels_to_text(train_dataset[0]['label'].tolist(), idx2char)}"')

      Example of text-to-label conversion "состава" -> [1, 79, 76, 79, 80, 62, 64, 62, 95]
      Example of label-to-text conversion [1, 79, 76, 79, 80, 62, 64, 62, 95] -> "состава"
```

Рисунок 8 – Пример преобразований меток в индексы и наоборот

Сначала был создан объект `Augmentor.Pipeline` для управления аугментацией изображений (рис. 9). К нему был добавлен сдвиг (`shear`) с максимальным сдвигом влево/вправо на 2 пикселя. Это преобразование применялось в 70% случаев.

Затем была добавлена операция случайного искажения (`random_distortion`). Изображение делилось на сетку 3x3, и каждый фрагмент сетки случайным образом искажался со смещением до 11 пикселей. Данная операция применялась в 100% случаев.

```
[50]: # Create an Augmentor pipeline object to manage image augmentations
p = Augmentor.Pipeline()

# Add a shear transformation with up to 2 pixels left/right shear
# Apply this shear 70% of the time
p.shear(max_shear_left=2, max_shear_right=2, probability=0.7)

# Add a random distortion transformation
# Divide image into 3x3 grid and distort each section randomly
# Apply magnitude of up to 11 pixels displacement
# Apply this distortion 100% of the time
p.random_distortion(probability=1.0, grid_width=3, grid_height=3, magnitude=11)
```

Рисунок 9 – Преобразования с помощью Augmentor.Pipeline

Следующим шагом стало создание композиции преобразований (`transforms.Compose`) для обучающей и тестовой выборок (рис. 10, 11). Для обучающей выборки были определены следующие операции: конвертация тензора в PIL изображение, преобразование в оттенки серого (1 канал), применение пайплайна Augmentor (искажение и сдвиг), случайное изменение контраста и насыщенности, случайное вращение в диапазоне от -9 до +9 градусов, случайное аффинное преобразование с масштабированием и сдвигом, а также конвертация обратно в тензор.

Для тестовой выборки использовалась более простая композиция: конвертация в PIL изображение, преобразование в оттенки серого и конвертация обратно в тензор.

```
[51]: channels = 1

train_transforms = transforms.Compose([
    # Convert tensor to PIL image
    transforms.ToPILImage(),

    # Convert images to 1 channel grayscale
    transforms.Grayscale(channels),

    # Apply Augmentor pipeline distortions
    p.torch_transform(), # random distortion and shear

    # Randomly change contrast and saturation
    transforms.ColorJitter(contrast=(0.5,1),saturation=(0.5,1)),

    # Randomly rotate between -9 and +9 degrees
    transforms.RandomRotation(degrees=(-9, 9)),

    # Random affine transformation with scaling and shear
    transforms.RandomAffine(10, None, [0.6, 1], 3, fill=255),

    # Random Gaussian blurring
    transforms.transforms.GaussianBlur(3, sigma=(0.1, 1.9)),

    # Convert PIL image to tensor
    transforms.ToTensor()
])


# Test time augmentation only converts PIL->tensor
test_transforms = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Grayscale(channels),
    transforms.ToTensor()
])
```

Рисунок 10 – Преобразования с помощью torch transforms; изображения в обучающей выборке будут немного изменяться каждую эпоху обучения, что обеспечивает аугментацию данных

```
[67]: transformed_img_ = train_transforms(processed_img_.astype(float))

[74]: print('Train transforms result (augmentaions enabled):')
      print(transformed_img_)
      transforms.ToPILImage()(transformed_img_)

      Train transforms result (augmentaions enabled):
      tensor([[[[1., 1., 1., ..., 1., 1., 1.],
                  [1., 1., 1., ..., 1., 1., 1.],
                  [1., 1., 1., ..., 1., 1., 1.],
                  ...,
                  [1., 1., 1., ..., 1., 1., 1.],
                  [1., 1., 1., ..., 1., 1., 1.],
                  [1., 1., 1., ..., 1., 1., 1.]])]])

[74]: 
```

```
[75]: transformed_test_img_ = test_transforms(processed_img_.astype(float))

[76]: print('Train transforms result (augmentaions disabled):')
      transforms.ToPILImage()(transformed_test_img_)


      Train transforms result (augmentaions disabled):
[76]: 
```

Рисунок 11 – Примеры преобразований изображений с аугментацией и без нее

Далее был реализован класс TextDataset для передачи данных в PyTorch (рис. 12).

```
[59]: # Create 3 Datasets
train_dataset = TextDataset(X_train, y_train, train_transforms, char2idx, idx2char)
val_dataset = TextDataset(X_val, y_val, test_transforms, char2idx, idx2char)
test_dataset = TextDataset(X_test, y_test, test_transforms, char2idx, idx2char)

[60]: # Print Datasets statistics

print('Train DS statistics:')
train_dataset.get_statistics()
print()

print('Val DS statistics:')
val_dataset.get_statistics()
print()

print('Test DS statistics:')
test_dataset.get_statistics()

Train DS statistics:
Dataset size = 7068
Longest text = 38 chars
Most common char = o (5394)
Least common char = » (1)

Val DS statistics:
Dataset size = 1591
Longest text = 32 chars
Most common char = o (1242)
Least common char = 4 (1)

Test DS statistics:
Dataset size = 1544
Longest text = 22 chars
Most common char = o (1321)
Least common char = Й (1)
```

Рисунок 12 – Создание трех экземпляров TextDataset, соответствующих выборкам

Далее была реализована функция подготовки батчей TextCollate (рис. 13, 14). Каждая последовательность дополнялась до максимальной длины в этом батче.

```
[78]: def TextCollate(batch):
    """
    Collates batches of images and text labels, padding the labels to a fixed length.
    This allows batches with variable length labels to be used for model training.

    Args:
        batch : List[Dict]
            A batch of examples, each containing an 'image' and 'label' field.

    Returns:
        Dict
            A batch compatible for model training:
            'images': Tensor of stacked images
            'labels': LongTensor of padded labels,
                    padded to the longest label length in the batch
    """
    # Extract images from the batch
    images = [b['image'] for b in batch]

    # Stack them into a tensor
    images = torch.stack(images)

    # Extract text labels from the batch
    labels = [b['label'] for b in batch]

    # Find the maximum label length in the batch
    max_batch_lenght = max(len(l) for l in labels)

    # Create an empty padded tensor for the labels
    padded_labels = torch.LongTensor(max_batch_lenght, len(batch))

    # Initialize it to all zeros
    padded_labels.zero_()

    # Copy the labels to the corresponding rows of the padded tensor
    for i in range(len(batch)):
        txt = labels[i]
        padded_labels[:txt.size(0), i] = txt

    return {
        'images': images,
        'labels': padded_labels
    }
```

Рисунок 13 – Функция подготовки батча; реализован паддинг до максимальной длины меток

```
[82]: print('Example of labels batch (tensor [seq_len, batch_size]):')
      next(iter(train_loader))['labels']

      Example of labels batch (tensor [seq_len, batch_size]):
[82]: tensor([[ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1],
              [75,  6, 64, 77, 37, 77, 76, 76, 85, 64, 78, 80, 77, 80, 77, 54],
              [81, 65, 75, 67, 76, 73, 80, 63, 67, 76, 67, 67, 78, 76, 76, 78],
              [68, 67, 62, 78, 70, 76, 75, 83, 73, 76, 65, 79, 70, 64, 78, 62],
              [75, 75, 93, 70,  9, 80, 76, 76, 76, 78, 70, 72, 79, 62,  9, 74],
              [76, 67,  2, 76, 95, 75, 79, 66, 64, 81, 76, 70, 93, 78, 72, 95],
              [95, 78, 82, 66,  0, 76, 93, 70, 67, 95, 75, 95, 65, 70, 62,  0],
              [ 0, 62, 76, 67,  0, 71, 80, 80, 72,  0, 95,  0, 67, 87, 95,  0],
              [ 0, 80, 78, 95,  0, 95, 79, 95, 62,  0,  0,  0, 95, 62,  0,  0],
              [ 0, 76, 74,  0,  0,  0, 93,  0, 95,  0,  0,  0,  0, 83,  0,  0],
              [ 0, 95, 62,  0,  0,  0, 95,  0,  0,  0,  0,  0,  0, 95,  0,  0],
              [ 0,  0, 95,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]])
```

Рисунок 14 – Пример подготовленного батча с метками

2 Архитектура модели

Архитектура модели представляет собой гибридную нейронную сеть, объединяющую сверточные слои и трансформер. Она наследуется от класса `nn.Module` из PyTorch.

Модель инициализируется (рис. 15) с указанием размера выходного словаря токенов, скрытого размера, а также количества слоев в энкодере, декодере и числа голов внимания. Также задается дропаут.

```
[111]: class TransformerModel(nn.Module):
    def __init__(self, out_token_size, hidden_size, enc_layers=1, dec_layers=1, nhead=1, dropout=0.1):
        super(TransformerModel, self).__init__()

        # Num of transformer layers
        self.enc_layers = enc_layers
        self.dec_layers = dec_layers

        # Initialize convolutional layers, batch normalization, and max pooling layers
        self.conv_layers, self.batch_norm_layers, self.pool_layers = self._initialize_layers(hidden_size)

        # Initialize activation function
        self.activation = nn.LeakyReLU()

        # Initialize positional encodings for src and tgt
        self.encoder_pe = PositionalEncoding(hidden_size, dropout)
        self.decoder_pe = PositionalEncoding(hidden_size, dropout)

        # Initialize character embedding for labels
        self.embedding = nn.Embedding(out_token_size, hidden_size)

        # Initialize transformer
        self.transformer = nn.Transformer(
            d_model=hidden_size,
            nhead=nhead,
            num_encoder_layers=enc_layers,
            num_decoder_layers=dec_layers,
            dim_feedforward=hidden_size * 4,
            dropout=dropout
        )

        # Initialize fully connected output layer
        self.fc_out = nn.Linear(hidden_size, out_token_size)

        # Initialize masks
        self.tgt_mask = None

        # Log configuration
        log_model_config(self)
```

Рисунок 15 – Инициализация модели

В конструкторе `init` инициализируются сверточные слои, слои батч-нормализации и максимального пулинга с помощью метода `_initialize_layers` (рис. 16). Здесь определяется архитектура сверточной части сети. Также создаются слои позиционного кодирования для энкодера и декодера, слой эмбеддингов для символов, трансформер и полносвязный выходной слой.

```

def _initialize_layers(self, hidden_size):
    """
    Initialize CNN layers

    Args:
        hidden_size : int

    Returns:
        conv_layers : nn.ModuleList, batch_norm_layers : nn.ModuleList, pool_layers : nn.ModuleList
    """
    # Define convolutional layers
    conv_layers = nn.ModuleList([
        nn.Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)),
        nn.Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)),
        nn.Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 1), padding=(1, 1)),
        nn.Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)),
        nn.Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 1), padding=(1, 1)),
        nn.Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)),
        nn.Conv2d(512, hidden_size, kernel_size=(2, 1), stride=(1, 1))
    ])

    # Define batch normalization layers
    batch_norm_layers = nn.ModuleList([
        nn.BatchNorm2d(64),
        nn.BatchNorm2d(128),
        nn.BatchNorm2d(256),
        nn.BatchNorm2d(256),
        nn.BatchNorm2d(512),
        nn.BatchNorm2d(512),
        nn.BatchNorm2d(hidden_size)
    ])

    # Define max pooling layers
    pool_layers = nn.ModuleList([
        None, # No max pooling after the zero convolutional layer
        nn.MaxPool2d(kernel_size=2, stride=2, padding=0),
        None, # No max pooling after the second convolutional layer
        nn.MaxPool2d(kernel_size=2, stride=2, padding=0),
        None, # No max pooling after the fourth convolutional layer
        nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 1), padding=(0, 1)),
        None # No max pooling after the sixth convolutional layer
    ])

    return conv_layers, batch_norm_layers, pool_layers

```

Рисунок 16 – Создание сверточных слоев для извлечения признаков

Метод `_get_features` (рис. 17) применяет сверточные слои, батч-нормализацию, активационную функцию и пулинг к входному изображению для извлечения признаков.

```
def _get_features(self, x):
    """
    Extracts features from the input using convolutional layers

    Args:
        src : Tensor [B, C, H, W]
            B - batch size, C - num of channels, H - height, W - width

    Returns:
        x : Tensor : [W, B, C*H]
            B - batch size, C - num of channels, H - height, W - width
    """
    for conv, bn, pool in zip(self.conv_layers, self.batch_norm_layers, self.pool_layers):
        x = self.activation(bn(conv(x)))
        if pool is not None:
            x = pool(x)
    """
    # [B, C, H, W] -> [B, W, C, H] -> [B, W, C*H] -> [W, B, C*H]
    # [16, 512, 1, 65] -> [16, 65, 1, 512] -> [16, 65, 512] -> [65, 16, 512] (seq_len, batch_size, hidden_size)
    x = x.permute(0, 3, 1, 2).flatten(2).permute(1, 0, 2)
    """

    # [B, C, H, W] -> [B, W, C*H] -> [W, B, C*H]
    # [16, 512, 1, 65] -> [16, 65, 512] -> [65, 16, 512] (seq_len, batch_size, hidden_size)
    x = x.flatten(2).permute(2, 0, 1)
    return x
```

Рисунок 17 – Извлечение признаков из изображения

Метод forward (рис. 18) реализует прямой проход модели. Сначала извлекаются признаки из изображения с помощью сверточных слоев. Затем применяется позиционное кодирование к извлеченным признакам. Для текстовых меток создается маска для обработки концов последовательностей, применяется эмбединг и позиционное кодирование. Далее извлеченные признаки и закодированные метки подаются на вход трансформера. Выход трансформера обрабатывается полносвязным слоем для получения индексов символов.

```
def forward(self, src, tgt):
    """
    Model's forward pass

    Args:
        src : Tensor [B, C, H, W]
            B - batch size, C - num of channels, H - height, W - width
        tgt : Tensor [L, B]
            L - max label length (tgt_len), B - batch size

    Returns:
        output : Tensor [L, B, O]
            L - max label length (tgt_len), B - batch, O - output token size
    """
    # 1 Images (src)

    # Extract features from images using Conv Layers
    # [batch_size, channels, height, width] -> [seq_len, batch_size, hidden_size]
    features = self._get_features(src)

    # Apply positional encoding to extracted features (before encoder's input)
    pos_encoded_features = self.encoder_pe(features)

    # 2 Labels (tgt)

    # Generates a square matrix where the each row allows one character more to be seen
    # The masked positions are filled with float('-inf')
    # Unmasked positions are filled with float(0.0).
    if self.tgt_mask is None or self.tgt_mask.size(0) != len(tgt):
        self.tgt_mask = nn.Transformer.generate_square_subsequent_mask(len(tgt)).to(tgt.device)

    # Create tgt_key_padding_mask (shape [batch_size, tgt_len])
    tgt_pad_mask = self.make_len_mask(tgt)

    # Apply word embedding to Labels (characters indexes)
    # [seq_len, batch_size] -> [seq_len, batch_size, hidden_size]
    embedded_tgt = self.embedding(tgt)

    # Apply positional encoding to characters embeddings
    pos_encoded_embedded_tgt = self.decoder_pe(embedded_tgt)

    # Transformer pass
    output = self.transformer(
        pos_encoded_features,
        pos_encoded_embedded_tgt,
        tgt_mask=self.tgt_mask,
        tgt_key_padding_mask=tgt_pad_mask,
    )

    # Final fully connected layer that converts transformer output to characters indexes
    output = self.fc_out(output)

    return output
```

Рисунок 18 – Прямой проход модели

Метод predict (рис. 19) используется для предсказания последовательности индексов символов по входному изображению. Он итеративно генерирует последовательность, применяя энкодер трансформера к извлеченным признакам, а затем декодер - к сгенерированной на предыдущем шаге последовательности индексов.

```
def predict(self, batch):
    """
    Method to predict sequences of token indexes based on input data (images) batch.

    Args:
        batch : Tensor [B, C, H, W]
                B - batch, C - channel, H - height, W - width

    Returns:
        result : List [B, L]
                Predicted sequences of token indexes
                B - batch size, L - max label length (tgt_len)
    """
    result = []
    for i, item in enumerate(batch):
        # Get features from the image
        features = self._get_features(item.unsqueeze(0))
        # Apply position encoding to the features
        pos_encoded_features = self.encoder_pe(features)
        # Apply transformer encoder to the encoded features
        memory = self.transformer.encoder(pos_encoded_features)

        # Initialize the list of indexes with the index of the start token
        out_indexes = [alphabet.index('_SOS'), ]

        # Generate the sequence of tokens
        for _ in range(100):
            # Convert the list of indexes to a tensor and add a batch dimension
            tgt_tensor = torch.LongTensor(out_indexes).unsqueeze(1).to(item.device)
            # Apply transformer decoder to the tensor and memory
            output = self.transformer.decoder(self.decoder_pe(self.embedding(tgt_tensor)), memory)
            # Apply the fully connected layer to the decoder output
            fc_output = self.fc_out(output)
            # Select the index of the token with the highest probability
            out_token = fc_output.argmax(2)[-1].item()
            # Add the token index to the list
            out_indexes.append(out_token)
            # If the end token is reached, break the loop
            if out_token == alphabet.index('_EOS'):
                # print(i)
                break

        # Add the predicted sequence of tokens to the result
        result.append(out_indexes)

    return result
```

Рисунок 19 – Метод для предсказания текста на изображении

На рис. 20 представлены выбранные параметры модели и ее создание.

```
[64]: # Model parameters
hidden_size = 512
enc_layers = 2
dec_layers = 2
n_heads = 4
dropout = 0.2

[113]: set_seed(1234)

[114]: # Initialize model
model = TransformerModel(
    len(alphabet),
    hidden_size=hidden_size,
    enc_layers=enc_layers,
    dec_layers=dec_layers,
    nhead=n_heads,
    dropout=dropout
)

Number of transformer encoder layers: 2
Number of attention heads: 4
Decoder embedding dimensionality: 512
Number of classes in output layer: 96
Dropout probability: 0.2
Number of trainable parameters: 19,842,274
TransformerModel(
  (conv_layers): ModuleList(
    (0): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (2): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 1), padding=(1, 1))
    (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 1), padding=(1, 1))
    (5): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): Conv2d(512, 512, kernel_size=(2, 1), stride=(1, 1))
  )
  (batch_norm_layers): ModuleList(
    (0): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

Рисунок 20 – Создание модели

3 Обучение и тестирование модели

3.1 Обучение

Для обучения модели были заданы следующие гиперпараметры (рис. 21): общее количество эпох - 200, параметр ранней остановки (early stopping patience) - 15, базовый коэффициент обучения (learning rate) - $1e-4$, размер батча - 128. Кроме того, было определено, что каждые 5 эпох будет сохраняться контрольная точка (checkpoint) для возможности продолжения обучения.

```
Parameters

[88]: # Total number of epochs
      n_epochs = 200

      # Early stopping patience
      patience = 15

      # Base LR
      learning_rate = 1e-4

[72]: # CP frequency
      checkpoint_n_epochs = 5

      # Models path
      models_path = f'{work_dir_path}models/'

      # CP path
      checkpoint_path = f'{models_path}checkpoints/'

[90]: optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
      criterion = nn.CrossEntropyLoss(ignore_index=char2idx['_PAD'])

[91]: # LR scheduler patience
      lr_reduce_patience = 7

      # Create LR scheduler (ReduceLRonPlateau)
      scheduler = torch.optim.lr_scheduler.ReduceLRonPlateau(optimizer, patience=lr_reduce_patience)

[70]: device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

Рисунок 21 – Гиперпараметры обучения модели

В качестве оптимизатора использовался Adam, а функцией потерь - кросс энтропия. Помимо базового коэффициента обучения, был задействован планировщик ReduceLRonPlateau для уменьшения коэффициента обучения с параметром patience, равным 7 эпохам.

Из-за высокой вычислительной сложности задачи и ограниченных ресурсов, обучение модели проводилось в несколько этапов по несколько эпох за раз с сохранением контрольных точек. Процесс обучения начинался с последней сохраненной контрольной точки, что позволяло продолжать работу, не теряя достигнутого прогресса.

После 95 эпох обучения модель достигла значения функции потерь 0.29 на обучающей выборке и 0.43 на валидационной выборке (рис. 22). Эти результаты демонстрируют способность модели обучаться на данных, однако для достижения более высокой производительности возможно потребуются дальнейшее обучение в течение большего количества эпох или оптимизация гиперпараметров и архитектуры модели.

Epoch 85/200:	Train loss: 0.37226699969985266;	Val loss: 0.46186662713686627;	LR: 0.0001
19% ■■■	26/140 [34:39<2:31:55, 79.96s/it]		
Epoch 86/200:	Train loss: 0.36020057472315703;	Val loss: 0.49096651871999103;	LR: 0.0001
19% ■■■	27/140 [35:58<2:30:16, 79.80s/it]		
Epoch 87/200:	Train loss: 0.3630861905488101;	Val loss: 0.47870177527268726;	LR: 0.0001
20% ■■■	28/140 [37:17<2:28:24, 79.50s/it]		
Epoch 88/200:	Train loss: 0.3538866487416354;	Val loss: 0.4746709292133649;	LR: 0.0001
21% ■■■	29/140 [38:38<2:27:38, 79.81s/it]		
Epoch 89/200:	Train loss: 0.34912806424227627;	Val loss: 0.4801930810014407;	LR: 0.0001
Epoch 90/200:	Train loss: 0.35097094720060174;	Val loss: 0.46998437742392224;	LR: 0.0001
22% ■■■	31/140 [41:18<2:25:04, 79.86s/it]		
Epoch 91/200:	Train loss: 0.33935282880609685;	Val loss: 0.4694041684269905;	LR: 0.0001
23% ■■■	32/140 [42:38<2:23:52, 79.93s/it]		
Epoch 92/200:	Train loss: 0.3392522242936221;	Val loss: 0.4851485565304756;	LR: 0.0001
24% ■■■	33/140 [43:58<2:22:29, 79.90s/it]		
Epoch 93/200:	Train loss: 0.32404449690472;	Val loss: 0.4721083144346873;	LR: 0.0001
24% ■■■	34/140 [45:17<2:20:45, 79.67s/it]		
Epoch 94/200:	Train loss: 0.2929695519534024;	Val loss: 0.43724258492390317;	LR: 1e-05

Рисунок 22 – Окончание обучения модели

3.2 Тестирование

Для оценки производительности обученной модели на тестовой выборке были реализованы две популярные метрики: символьная ошибка (Character Error Rate, CER) и ошибка слов (Word Error Rate, WER) (рис. 23). Эти метрики широко используются для измерения точности систем распознавания текста.

```
[84]: def character_error_rate(sequence1, sequence2):
    """
    Calculate the Character Error Rate (CER) between two sequences of characters.

    Args:
        sequence1 : str
            The first sequence of characters.
        sequence2 : str
            The second sequence of characters.

    Returns:
        cer : float
            The Character Error Rate (CER) between the two sequences.
    """
    # Create a set of unique characters from both sequences
    vocabulary = set(sequence1 + sequence2)

    # Create a dictionary that maps each unique character to a unique integer
    char_to_index = dict(zip(vocabulary, range(len(vocabulary))))

    # Convert the sequences to lists of characters using the mapping
    char_sequence1 = [chr(char_to_index[char]) for char in sequence1]
    char_sequence2 = [chr(char_to_index[char]) for char in sequence2]

    # Calculate the distance between the two sequences
    distance = editdistance.eval(''.join(char_sequence1), ''.join(char_sequence2))

    # Calculate the Character Error Rate as the ratio of the distance to the maximum length of the two sequences
    cer = distance / max(len(sequence1), len(sequence2))

    return cer

[85]: print(f'CER between "hello" and "world" is {character_error_rate("hello", "world")}')

CER between "hello" and "world" is 0.8
```

Рисунок 23 – Функция вычисления CER

На тестовой выборке значение CER составило 0.282, а WER - 0.774 (рис. 24). Такие результаты указывают на довольно высокий уровень ошибок при распознавании отдельных символов и целых слов соответственно. Однако важно отметить, что данные метрики были получены на ограниченном наборе тестовых данных и после относительно небольшого количества эпох обучения.

```
[126]: # Calculate mean values
test_cer_score = np.mean(test_cer_scores)
test_wer_score = np.mean(test_wer_scores)

# Print results
print(f'Test mean CER: {test_cer_score}; Test mean WER: {test_wer_score}')

Test CER: 0.2820423270225539; Test WER: 0.7740885416666666
```

Рисунок 24 – Вычисленные значения метрик

Кроме того, для удобства работы с моделью была реализована функция `make_prediction`, которая позволяет получать предсказание для одного входного изображения (рис. 25, 26). Эта функция предназначена для визуализации результатов работы модели и проверки ее способности распознавать рукописный текст на отдельных примерах.

```
[74]: def make_prediction(model, img, idx2char):
    """
    Recognize text on single image

    Args:
        model : nn.Module
            Model using to make prediction
        img : numpy.ndarray
            Image in array format
        char2idx : dict
            Mapping chars to indexes
        idx2char : dict
            Mapping indexes to chars

    Returns:
        predicted_text : str
            Predicted text for the image
    """

    # Load image and apply necessary transformations
    img = process_image(img)
    img = img / img.max()
    img = np.transpose(img, (2, 0, 1))
    img = torch.FloatTensor(img).unsqueeze(0)
    img = transforms.Grayscale(1)(img)
    img = img.to(device)

    # Make predictions using model
    model.to(device)
    model.eval()
    with torch.no_grad():
        output_indexes = model.predict(img)

    # Convert indexes to text
    predicted_text = labels_to_text(output_indexes[0], idx2char)

    return predicted_text
```

Рисунок 25 – Функция для предсказания текста на одном изображении

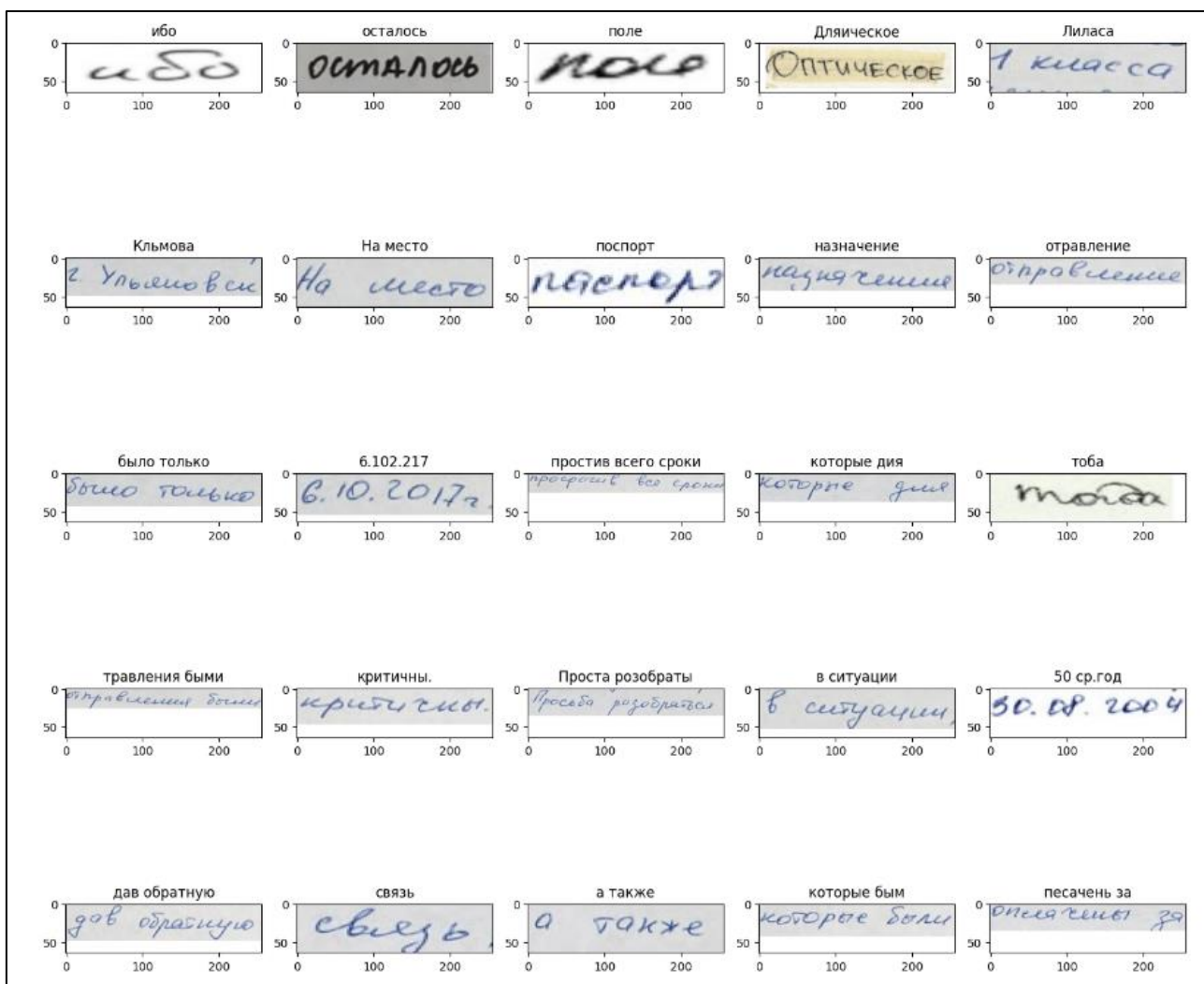


Рисунок 26 – Примеры предсказаний модели

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной курсовой работы были успешно решены все поставленные задачи.

- 1) Был собран и подготовлен набор данных из 10204 примеров с образцами рукописного кириллического текста для задачи распознавания. Выполнена предобработка данных, которая включала фильтрацию нежелательных символов, нормализацию размеров изображений до 64x256 пикселей, преобразование меток в индексы и применение различных методов аугментации изображений. Это позволило подготовить качественные данные для обучения нейронной сети.
- 2) Разработана гибридная архитектура нейронной сети, объединяющая 5 сверточных слоев для извлечения визуальных признаков и трансформер в энкодере и декодере для моделирования контекстных зависимостей. Использование такого подхода позволило эффективно обрабатывать пространственную информацию из изображений и учитывать последовательную природу текстовых данных.
- 3) Проведено обучение разработанной нейронной сети на подготовленном наборе данных в течение 95 эпох (значение функции потерь на обучающей выборке составило 0.29, а на валидационной 0.43). В процессе обучения применялись методы регуляризации, оптимизации гиперпараметров и сохранение контрольных точек для возможности дальнейшего продолжения. Обученная модель была протестирована на отдельной тестовой выборке с использованием метрик символьной ошибки (CER = 0.282) и ошибки слов (WER = 0.774). Полученные результаты демонстрируют способность модели распознавать рукописный кириллический текст, но также указывают на возможность дальнейшего улучшения производительности.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. pandas documentation — pandas 2.2.1 documentation. — URL: <https://pandas.pydata.org/docs/> (дата обращения: 27.03.2024). — Текст : электронный.
2. PyTorch documentation — PyTorch 2.2 documentation. — URL: <https://pytorch.org/docs/stable/index.html> (дата обращения: 27.03.2024). — Текст : электронный.
3. Transformer Neural Networks, ChatGPT's foundation, Clearly Explained!!! — 2023. — URL: <https://www.youtube.com/watch?v=zxQyTK8quyY> (дата обращения: 27.03.2024). — Текст : электронный.
4. Verner, K. konverner/shiftlab_ocr / K. Verner. — 2024. — URL: https://github.com/konverner/shiftlab_ocr (дата обращения: 27.03.2024). — Текст : электронный.
5. Word Embedding and Word2Vec, Clearly Explained!!! — 2023. — URL: <https://www.youtube.com/watch?v=viZrOnJclY0> (дата обращения: 27.03.2024). — Текст : электронный.
6. 4. Character Error Rate and Learning Curve. — URL: <https://help.transkribus.org/character-error-rate-and-learning-curve> (date accessed: 27.03.2024). — Text : electronic.
7. Convolutional neural network / Text : electronic // Wikipedia / Page Version ID: 1214849746. — 2024. — URL: https://en.wikipedia.org/w/index.php?title=Convolutional_neural_network&oldid=1214849746 (date accessed: 27.03.2024).
8. Cyrillic Handwriting Dataset. — URL: <https://www.kaggle.com/datasets/constantinwerner/cyrillic-handwriting-dataset> (date accessed: 27.03.2024). — Text : electronic.
9. Explore Cyrillic Handwriting Dataset. — URL: <https://kaggle.com/code/constantinwerner/explore-cyrillic-handwriting-dataset> (date accessed: 27.03.2024). — Text : electronic.
10. Phillips, H. Positional Encoding / H. Phillips. — 2023. — URL: <https://medium.com/@hunter-j-phillips/positional-encoding-7a93db4109e6> (date accessed: 27.03.2024). — Text : electronic.
11. Transformer (deep learning architecture) / Text : electronic // Wikipedia / Page Version ID: 1214873564. — 2024. — URL: [https://en.wikipedia.org/w/index.php?title=Transformer_\(deep_learning_architecture\)&oldid=1214873564](https://en.wikipedia.org/w/index.php?title=Transformer_(deep_learning_architecture)&oldid=1214873564) (date accessed: 27.03.2024).
12. WER, CER, and MER - Testing with Kolena. — URL: <https://docs.kolena.com/metrics/wer-cer-mer/> (date accessed: 27.03.2024). — Text : electronic.