Текст → Обработка → Encoder → Attention → Decoder → Mel → Аудио

1. Text Processor + Энкодер:

   Текст кодируется посимвольно:

```
class TextProcessor:
    def __init__(self, alphabet):
        self.char_to_id = {char: i + 1 for i, char in enumerate(alphabet)}
        self.id_to_char = {i + 1: char for i, char in enumerate(alphabet)}
        self.pad_id = 0

    def encode(self, text):
        text = text.lower()
        return [self.char_to_id[c] for c in text if c in self.char_to_id]

    def decode(self, ids):
        return "".join([self.id_to_char[i] for i in ids if i in
self.id_to_char])
```

   Энкодер:

- принимает последовательность символов
- преобразует их в embedding

На выходе мы получаем контекстное представление текста.

```
class Encoder(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(
            embedding_dim,
            hidden_dim,
            num_layers=2,
            batch_first=True,
            bidirectional=True
        )

    def forward(self, x, lengths):
        x = self.embedding(x)
        packed_x = nn.utils.rnn.pack_padded_sequence(
            x, lengths.cpu(), batch_first=True, enforce_sorted=False
        )
        packed_outputs, _ = self.lstm(packed_x)
        outputs, _ = nn.utils.rnn.pad_packed_sequence(packed_outputs,
batch_first=True)
        return outputs
```

3. Используется механизм Bahdanau Attention.

На каждом шаге декодирования:

- декодер формирует запрос
- attention вычисляет веса важности символов
- строится контекстный вектор

Это позволяет модели правильно выравнивать текст и аудио во времени.

```
class BahdanauAttention(nn.Module):
    def __init__(self, encoder_dim, decoder_dim, attention_dim):
        super().__init__()
        self.W1 = nn.Linear(encoder_dim, attention_dim, bias=False)
        self.W2 = nn.Linear(decoder_dim, attention_dim, bias=False)
        self.V = nn.Linear(attention_dim, 1, bias=False)

    def forward(self, query, keys, mask=None):
        proj_key = self.W1(keys)
        proj_query = self.W2(query)

        scores = self.V(torch.tanh(proj_key + proj_query)).squeeze(-1)

        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        weights = F.softmax(scores, dim=-1)
        context = torch.bmm(weights.unsqueeze(1), keys)
        return context, weights
```

4. Декодер:

- работает авторегрессионно
- на каждом шаге генерирует один mel-кадр
- использует attention-контекст

```
class Decoder(nn.Module):
    def __init__(self, n_mels, decoder_hidden, encoder_total_dim,
attention_dim):
        super().__init__()
        self.n_mels = n_mels
        self.decoder_hidden = decoder_hidden

        self.lstm = nn.LSTMCell(n_mels + encoder_total_dim, decoder_hidden)
        self.attention = BahdanauAttention(encoder_total_dim, decoder_hidden,
attention_dim)

        self.linear = nn.Linear(decoder_hidden + encoder_total_dim, n_mels)

        self.stop_linear = nn.Linear(decoder_hidden + encoder_total_dim, 1)

    def forward(self, encoder_outputs, encoder_mask, teacher_mels=None,
max_len=1000):
        batch_size = encoder_outputs.size(0)
        device = encoder_outputs.device

        h = torch.zeros(batch_size, self.decoder_hidden).to(device)
        c = torch.zeros(batch_size, self.decoder_hidden).to(device)
```

```
        mel_input = torch.zeros(batch_size, self.n_mels).to(device)

        outputs = []
        stop_tokens = []


        if teacher_mels is not None:
            steps = teacher_mels.size(1)
        else:
            steps = max_len

        for t in range(steps):
            context, _ = self.attention(h.unsqueeze(1), encoder_outputs,
encoder_mask)
            context = context.squeeze(1)

            rnn_input = torch.cat([mel_input, context], dim=-1)
            h, c = self.lstm(rnn_input, (h, c))

            concat_out = torch.cat([h, context], dim=-1)

            mel_out = self.linear(concat_out)               stop_out =
self.stop_linear(concat_out)

            outputs.append(mel_out)
            stop_tokens.append(stop_out)

            if teacher_mels is not None:

                if t < teacher_mels.size(1) - 1:
                    mel_input = teacher_mels[:, t, :]
                else:
                    mel_input = mel_out
            else:

                if torch.sigmoid(stop_out).item() > 0.5:
                    break
                mel_input = mel_out

        return torch.stack(outputs, dim=1), torch.stack(stop_tokens, dim=1)
```

5. Сама модель

Text → Encoder → Attention → Decoder → Mel + Stop

Модель возвращает:

- mel-спектрограмму
- вероятности остановки

```
class StudentTTS(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.encoder = Encoder(cfg.vocab_size, cfg.embedding_dim,
cfg.encoder_hidden)
        self.decoder = Decoder(cfg.n_mels, cfg.decoder_hidden,
cfg.encoder_hidden * 2, cfg.attention_dim)
```

```
    def forward(self, text, text_lengths, mels=None):
        device = text.device


        mask = torch.arange(text.size(1),
device=device).expand(len(text_lengths),
                                                                text.size(1))
< text_lengths.unsqueeze(1)

        encoder_outputs = self.encoder(text, text_lengths)

        mel_outputs, stop_outputs = self.decoder(encoder_outputs, mask,
teacher_mels=mels)

        return mel_outputs, stop_outputs
```

6. В обучении используется учитель — модель XTTS:

- студент учится не только на реальном аудио
- но и на "идеальной" спектрограмме учителя

Loss-функция комбинирует:

loss = alpha * MSE + beta * L1

7. Train loop

```
def train_with_distillation(root_dir):
    def masked_mse(preds, targets, mask):
        diff = (preds - targets) ** 2
        return (diff * mask).sum() / (mask.sum() * cfg.n_mels + 1e-8)

    def masked_l1(preds, targets, mask):
        diff = torch.abs(preds - targets)
        return (diff * mask).sum() / (mask.sum() * cfg.n_mels + 1e-8)


    cfg = Config()
    tp = TextProcessor(cfg.RUS_ALPHABET)
    writer = SummaryWriter(log_dir="runs/fast_distill_v2")
    student = StudentTTS(cfg).to(cfg.device)
    optimizer = torch.optim.AdamW(student.parameters(), lr=cfg.lr)

    bce_loss =
nn.BCEWithLogitsLoss(pos_weight=torch.tensor([5.0]).to(cfg.device))

    dataset = PodcastDistillDataset(root_dir, tp, cfg)
    dataloader = DataLoader(dataset, batch_size=cfg.batch_size,
collate_fn=collate_fn_podcast, shuffle=True)

    global_step = 0
    start_epoch = 0

    checkpoint_dir = "checkpoints"
    os.makedirs(checkpoint_dir, exist_ok=True)

    checkpoints = [f for f in os.listdir(checkpoint_dir) if
f.endswith('.pth') and 'step' in f]
```

```
    if checkpoints:
        checkpoints.sort(key=lambda x: int(x.split('_')[-
1].split('.')[0]))
        last_checkpoint = os.path.join(checkpoint_dir, checkpoints[-1])
        print(f"Загрузка чекпоинта: {last_checkpoint}")
        try:
            ckpt = torch.load(last_checkpoint)
            student.load_state_dict(ckpt['model_state_dict'])
            optimizer.load_state_dict(ckpt['optimizer_state_dict'])
            global_step = ckpt.get('global_step', 0)
            start_epoch = ckpt.get('epoch', 0)
            print(f"Восстановлено: Эпоха {start_epoch}, Шаг
{global_step}")
        except Exception as e:
            print(f"Ошибка загрузки чекпоинта, начинаем с нуля: {e}")

    print("Начало обучения с Stop Token...")

    student.train()

    try:
        for epoch in range(start_epoch, cfg.epochs):
            for batch in dataloader:

                tokens, token_lens, gts, gt_lens, raw_texts,
audio_paths = batch

                tokens = tokens.to(cfg.device)
                token_lens = token_lens.to(cfg.device)
                gts = gts.to(cfg.device)
                gt_lens = gt_lens.to(cfg.device)

                target_mels = gts

                optimizer.zero_grad()

                pred_mels, pred_stops = student(tokens, token_lens,
mels=target_mels)


                min_len = min(pred_mels.size(1), target_mels.size(1))

                p_mel = pred_mels[:, :min_len, :]
                t_mel = target_mels[:, :min_len, :]
                p_stop = pred_stops[:, :min_len, :]


                mask = torch.arange(min_len,
device=cfg.device).expand(len(gt_lens), min_len) < gt_lens.unsqueeze(1)
                mask_expanded = mask.unsqueeze(-1).float

                loss_mse = masked_mse(p_mel, t_mel, mask_expanded)
                loss_l1 = masked_l1(p_mel, t_mel, mask_expanded)

                stop_targets = torch.zeros_like(p_stop)

                for i, length in enumerate(gt_lens):

                    if length < min_len:
                        stop_targets[i, length:, 0] = 1.0

                loss_stop = bce_loss(p_stop, stop_targets)
```

```
                loss = (cfg.alpha * loss_mse) + (cfg.beta * loss_l1) +
loss_stop

                loss.backward()

                torch.nn.utils.clip_grad_norm_(student.parameters(),
1.0)

                optimizer.step()

                global_step += 1

                if global_step % 10 == 0:
                    writer.add_scalar('Loss/Total', loss.item(),
global_step)
                    writer.add_scalar('Loss/Mel_MSE', loss_mse.item(),
global_step)
                    writer.add_scalar('Loss/L1', loss_l1.item(),
global_step)
                    writer.add_scalar('Loss/Stop_BCE',
loss_stop.item(), global_step)
                    print(
                        f"Step {global_step} | Total: {loss.item():.4f}
| Mel: {loss_mse.item():.4f} | Stop: {loss_stop.item():.4f}")

                if global_step % 250 == 0:
                    save_path = os.path.join(checkpoint_dir,
f"student_step_{global_step}.pth")
                    torch.save({
                        'global_step': global_step,
                        'epoch': epoch,
                        'model_state_dict': student.state_dict(),
                        'optimizer_state_dict': optimizer.state_dict(),
                        'loss': loss.item(),
                    }, save_path)
                    print(f"💾 Чекпоинт сохранен: {save_path}")

    except KeyboardInterrupt:
        print("\nОстановка обучения пользователем...")
        save_path = os.path.join(checkpoint_dir, "interrupted.pth")
        torch.save({
            'global_step': global_step,
            'epoch': epoch,
            'model_state_dict': student.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss.item(),
        }, save_path)
        print("💾 Аварийный чекпоинт сохранен.")

    writer.close()
    print("Обучение завершено.")
```

8. Дальнейшие действия:

- Инференс модели посредством использования предобученного вокодера.
- Реализация клонирования голоса посредством speaker embedding