عنوان کارآموزی

گزارش کارآموزی

ارائه‌شده به:
دکتر رحمتی

توسط:
امیرمسعود شاکر

عنوان گروه
دانشکده مهندسی و علوم کامپیوتر
تابستان سال 1401

پیش‌گفتار

فهرست مطالب

چکیده

در این گزارش ابتدا به‌طور کلی به معرفی شرکت کارآموزی (پژوهشگاه دانش‌های بنیادی) پرداخته می‌شود. سپس فعالیت‌های مرتبط با حوزه‌ی علوم کامپیوتر این پژوهشگاه بررسی می‌شود. پس از آن، شرحی کلی از هدف و فرآیند کارآموزی ارائه می‌شود و در نهایت مفصلا به توضیح مراحل مختلف کارآموزی پرداخته می‌شود.

1. مقدمه

1.1. معرفی شرکت/سازمان محل کارآموزي

پژوهشگاه دانش‌های بنیادی مؤسسه‌ای وابسته به وزارت علوم، تحقیقات، و فناوری است که در سال ۱۳۶۸ با نام «مرکز تحقیقات فیزیک نظری و ریاضیات» تاسیس شد و هدف اولیه آن پیشبرد پژوهش و نوآوری در این دو رشته، و ضمناً فراهم آوردن الگویی بود که به ترویج و اعتلای فرهنگ پژوهش در سطح کشور کمک کند. مرکز فعالیت خود را با 3 هسته تحقیقاتی در فیزیک نظری و ۳ هسته تحقیقاتی در ریاضیات و با امکاناتی اندک آغاز کرد ولی به تدریج با توسعه امکانات و جذب دانشورانی از رشته‌های دیگر، فعالیت آن به حیطه‌های دیگری گسترش یافت و در سال ۱۳۷۶ نام آن به «پژوهشگاه دانش‌های بنیادی» تغییر کرد. این پژوهشگاه در حال حاضر با ۹ پژوهشکده در زمینه‌های گوناگون علوم بنیادی و برخورداری از زیرساخت‌ها و امکانات لازم (شبکه الکترونیکی، کامپیوتر، آزمایشگاه‌ها، و کتابخانه مجهز و روزآمد) که دایماً هم رو به توسعه است، حضور فعالی در جریان پژوهشی کشور در این دانش‌ها دارد. شرح وظایف و نوع فعالیت‌های پژوهشگاه در بخش آینده خواهد آمد، ولی در یک نگاه کلی به تجربهٔ چند دهه فعالیت پژوهشگاه، سه ویژگی بارز در این تجربه مشهود است: اول، کمیت و کیفیت تحقیقات انجام‌شده در این نهاد، یعنی کثرت تعداد مقاله‌های پژوهشی چاپ‌شده آن در مجله‌های علمی معتبر و تعداد استنادها به آنها؛ دوم، نوعی مدیریت پویا در امر پژوهش که می‌تواند الگویی برای مؤسسات تحقیقاتی باشد یعنی مدیریتی بر اساس «محوریت محقق»، «استقلال مدیریتی واحدهای پژوهشی»، و «انعطاف‌پذیری در تأسیس و انحلال آنها»؛ سوم، نگرش «ملی» پژوهشگاه که تقویت جریان کلی پژوهش در کشور مدّنظر داشته است. همکاری با مراکز پژوهشی دیگر و دانشگاه‌ها به صورت‌های مختلف و ارائه انواع تسهیلات به پژوهشگران آنها همواره جزیی از روال کار پژوهشگاه بوده است. ابتکار پژوهشگاه در ایجاد شبکه ارتباطی الکترونیکی (به نام شبکه علمی‌تحقیقاتی ایران یا ایرانت) در سال ۱۳۷۱، که به تدریج و برای اولین بار مراکز علمی‌تحقیقی و دانشگاه‌های ایران را به یکدیگر و به جهان علم در خارج مربوط ساخت، و اهتمام پژوهشگاه به اجرای طرح‌های ملی، مانند رصدخانه ملی ایران و شتابگر ملی، از جلوه‌ها و ثمرات این نگرش ملی است.

1.2. شرح کلی فعالیت شرکت/سازمان محل کارآموزي

- انجام تحقیقات در زمینه‌های مرتبط با موضوع تاسیس پژوهشگاه به طور مستقل و یا با همکاری مراکز علمی و پژوهشی داخل و خارج کشور؛
- ایجاد ارتباط فعال و سازنده با سایر مؤسسات و جوامع علمی و پژوهشی در داخل و خارج از کشور از طریق برگزاری انواع همایش‌ها، مبادله محقق و اجرای طرح‌های مشترک؛
- همکاری با دانشگاه‌ها و مراکز آموزش عالی و مؤسسات پژوهشی کشور و سایر نهادها در راستای پیشبرد اهداف موضوع تاسیس پژوهشگاه از طریق ارائه تسهیلات مختلف، پذیرش طرح‌های تحقیقاتی، ایجاد امکان گذراندن فرصت‌های مطالعاتی در پژوهشگاه؛
- ایجاد زمینه‌های مناسب برای جذب دانشمندان و پژوهشگران ایرانی؛
- کمک به پرورش محقق در زمینه‌های موضوع تاسیس از طریق دایر کردن دوره‌های تحصیلات تکمیلی و اعطا کمک هزینه تحصیلی؛
- نشر و ترویج یافته‌های علمی در زمینه‌های فعالیت پژوهشگاه از طریق انتشار کتب و نشریات و تشکیل تجمعات پژوهشی و آموزشی؛
- ارائه خدمات علمی و فنی در چارچوب فعالیت‌های پژوهشگاه؛
- بررسی و شناسایی نیازهای پژوهشی در زمینه دانش‌های بنیادی؛
- تاسیس مرکز خدمات شبکه‌ای با هدف برقراری ارتباطات شبکه‌ای جهت ارائه خدمات به پژوهشگاه و سایر مراکز علمی و پژوهشی و متقاضیان دیگر و همچنین تلاش برای توسعه فنون مربوط به شبکه در کشور.

1.3. شرح فعالیت‌هاي مرتبط شرکت/سازمان با رشته تحصیلی

به طور کلی **پژوهشگاه دانش های بنیادی** شامل پژوهشکده های زیر می باشد:

- پژوهشکده ذرات و شتابگرها
- پژوهشکده ریاضیات
- پژوهشکده علوم زیستی
- پژوهشکده علوم شناختی
- **پژوهشکده علوم کامپیوتر**
- پژوهشکده علوم نانو
- پژوهشکده فلسفه تحلیلی
- پژوهشکده فیزیک
- پژوهشکده نجوم

فعالیت من در این کارآموزی تحت نظارت پژوهشکده علوم کامپیوتر و به طول خاص **مرکز پردازش سریع *(High Performance Computing)*** بوده است. تمرکز اصلی این بخش، همانطور که از اسم آن مشخص است، بر روی بهبود سرعت و عملکرد بخش های محاسباتی است.

2. هدف از کارآموزي

هدف اصلی این کارآموزی توسعه‌ی یک کتابخانه‌ی بر روی زبان‌های سطح پایین بوده است که بتوان عملکرد مدل‌های مختلف را به لحاظ سخت افزاری مقایسه کرد. در دوره کارآموزی اینجانب، مدل های LSTM و GRU ابتدا به زبان c و سپس با استفاده از نرم افزار Vivado HLS به زبان‌های HDL تبدیل شد.

2.1. فعالیت‌هاي انجام شده

- مطالعه مدل LSTM
- جست و جو برای پیاده سازی مدل LSTM به زبان python و شروع پیاده سازی به زبان c به کمک کد python
- تکمیل پیاده سازی کد c و تست آن
- توضیح توابع پیاده سازی شده در بخش LSTM
- مطالعه مدل های Bidirectional RNN و پیاده سازی مدل Bi - LSTM به زبان c و تست آن
- مطالعه مدل GRU
- پیاده سازی مدل GRU به زبان c و تست آن
- توضیح توابع پیاده سازی شده در بخش GRU
- پیاده سازی مدل Bi - GRU به زبان c و تست آن
- سنتز کد های c به وسیله ابزار Vivado HLS

3. شرح فعالیت‌هاي کارآموزي

3.1. مطالعه مدل LSTM

در این مرحله یک مطالعه و مرور کلی روی مدل LSTM انجام شد.
مدل LSTM یک نوع خاص از مدل های (Recurrent Neural Network) RNN است که برای داده های دنباله ای استفاده میشود و برای بهبود عملکرد مدل های قدیمی RNN یا اصطلاحا Simple RNNs به وجود آمده است.
پس از مطالعه کلی، مراحل پیاده سازی این مدل به زبان c آغاز شد.

3.2. جست و جو برای پیاده سازی مدل LSTM به زبان python و شروع پیاده سازی به زبان c به کمک کد python

در آغاز مرحله پیاده سازی، پیاده سازی های موجود که اکثرا به زبان python بودند بررسی شدند تا با استفاده از آن، بتوان راحت تر پیاده سازی به زبان c را انجام داد.
پس از یافتن یک پیاده سازی مناسب به زبان python، انجام پیاده سازی به زبان c شروع شد.
در ابتدای کار، به دلیل محدودیت های فراوان زبان c و سطح پایین تر بودن آن نسبت به زبان python، انجام پیاده سازی به زبان c (در واقع تبدیل کد python به کد c) مقداری دشوار بود.
یکی از مشکلات، نبود مفهوم کلاس در زبان c بود که در پایتون وجود دارد و در واقع در کد python، تمام مدل در قالب یک کلاس پیاده سازی شده بود. از این رو، رویکرد functional programming در پیش گرفته شد.
همچنین چون بنده خیلی با زبان c سر و کار ندارم و آخرین بار ترم 1 با این زبان کد زده بودم، انجام پیاده سازی با آن در شروع کار کمی زمان بر بود.
پس از این طی این مرحله، پیاده سازی با سرعت بیشتری پیش رفت.

3.3. تکمیل پیاده سازی کد c و تست آن

بخش اصلی کار، یعنی پیاده سازی مدل LSTM در این مرحله انجام شد.
پس از اینکه در مرحله قبل زبان c مقداری مرور شد و بخش کوچکی از پیاده سازی نیز انجام شد، در این مرحله تمام قسمت های مدل LSTM پیاده سازی شد.
تفاوت های زبان های python , c در این مرحله بسیار به چشم آمد.
چرا که در زبان python بسیاری از عملیات لازم و ضروری، با یک خط کد و حتی با یک علامت انجام میشوند اما در زبان c انجام همان کار ممکن است نیاز به نوشتن یک تابع داشته باشد.
به طور مثال انجام عملیات جمع، تفریق، ضرب و تقسیم روی ماتریس ها در زبان python با استفاده از علامت این عملیات انجام میشود اما در زبان c برای هر کدام از این عملیات، نیاز به یک تابع مجزا داریم که دو آرایه ی دو بعدی ورودی بگیرد و یک آرایه را خروجی دهد.
نکته دیگر وجود توابع built in و کتابخانه های فراوان در زبان python است که کار را بسیار ساده میکند اما چنین امکاناتی در زبان c موجود نیست.
در نهایت پس از تکمیل پیاده سازی، به جهت تست کد پیاده سازی شده، کد python ای که پیاده سازی از روی آن انجام شد، کمی تغییر داده شد تا بتوان مشابه با کد c به آن ورودی داد و خروجی گرفت.
ورودی های یکسان به کد های c, python به ازای hyper parameter های یکسان داده شد و خروجی های یکسان گرفته شد.
در نتیجه صحت پیاده سازی c در اینجا تایید شد.
سپس باید این مدل به حالت Bidirectional تبدیل میشد.

3.4. توضیح توابع پیاده سازی شده در بخش LSTM

در این قسمت توابع پیاده سازی شده به زبان c توضیح داده میشوند.
- ابتدا توابع فایل utils.h توضیح داده میشوند.
این توابع در هر دو قسمت LSTM و GRU استفاده میشوند.

```
// initializes a matrix with shape: (row1, column1)
// used to initialize W_i, W_f, W_c, W_o and U_i, U_f, U_c,
U_o Matrices
void initialize_matrix(double** matrix, int row, int column,
double initial_weight);
```

تابع initialize_matrix در ورودی آرایه دو بعدی matrix به همراه سطر و ستون آن و عدد اعشاری initial_weight را میگیرد و تمام عناصر matrix را با مقدار initial_weight پر میکند.
این تابع برای مقدار دهی اولیه ماتریس های W_i, W_f, W_c, W_o و U_i, U_f, U_c, U_o استفاده میشود.

```
// initializes a vector with size: vec_size
// used to initialize b_i, b_f, b_c, b_o vectors
void initialize_vector(double* vector, int vec_size, double
initial_weight);
```

تابع initialize_vector در ورودی آرایه vector به همراه اندازه آن و عدد اعشاری initial_weight را میگیرد و تمام عناصر vector را با مقدار initial_weight پر میکند.
این تابع برای مقدار دهی اولیه وکتور های b_i, b_f, b_c, b_o استفاده میشود.

```
// multiplies two matrices together
// mat1 shape: (row1, column1)
// mat2 shape: (row2, column2)
double** matrix_mult(double** mat1, int row1, int column1,
double** mat2, int row2, int column2);
```

تابع matrix_mult در ورودی آرایه های دوبعدی mat1, mat2 به همراه سطر و ستون آنها را میگیرد و حاصل ضرب شان را خروجی میدهد.

```
// sum of two matrices with shape: (row, column)
double** matrix_sum_2(double** matrix1, double** matrix2, int
row, int column);
```

تابع matrix_sum_2 در ورودی آرایه های دوبعدی matrix1, matrix2 به همراه سطر و ستون آنها را میگیرد و حاصل جمع شان را خروجی میدهد.

```
// sum of three matrices with shape: (row, column)
double** matrix_sum_3(double** matrix1, double** matrix2,
double** matrix3, int row, int column);
```

تابع matrix_sum_3 در ورودی آرایه های دوبعدی matrix1, matrix2, matrix3 به همراه سطر و ستون آنها را میگیرد و حاصل جمع شان را خروجی میدهد.

```
// broadcast a vector with size vec_size to a matrix with
shape: (row, vec_size)
double** broadcast_vector_to_matrix(double* vector, int row,
int vec_size);
```

تابع broadcast_vector_to_matrix در ورودی آرایه vector به همراه اندازه آن (vec_size) و عدد صحیح row را ورودی میگیرد و یک آرایه دوبعدی با ابعاد (row, vec_size) برمیگرداند که گسترش یافته آرایه vector است.

```
// sigmoid function for a number
double sigmoid(double n);
```

تابع sigmoid در ورودی یک عدد اعشاری میگیرد و حاصل اعمال تابع ریاضی sigmoid را خروجی میدهد.

```
// sigmoid function broadcasted to a matrix with shape:
(batch_size, hidden_size)
double** matrix_sigmoid(double** matrix, int row, int column);
```

تابع matrix_sigmoid در ورودی آرایه دوبعدی matrix و سطر و ستون آن را میگیرد و تابع sigmoid را روی همه ی عناصر آن اعمال میکند و خروجی میدهد.

```
// tanh function broadcasted to a matrix
double** matrix_tanh(double** matrix, int row, int column);
```

تابع matrix_tanh در ورودی آرایه دوبعدی matrix و سطر و ستون آن را میگیرد و تابع tanh را روی همه ی عناصر آن اعمال میکند و خروجی میدهد.

```
// matrix product of two matrices with shape: (batch_size,
hidden_size)
double** matrix_product(double** matrix1, double** matrix2,
int row, int column);
```

تابع matrix_product در ورودی آرایه های دوبعدی matrix1, matrix2 به همراه سطر و ستون آنها را میگیرد و حاصل ضرب نقطه ای آنها را خروجی میدهد.

```
// average of two matrices with shape: (row, column)
double** matrix_avg_2(double** matrix1, double** matrix2, int
row, int column);
```

تابع matrix_avg_2 در ورودی آرایه های دوبعدی matrix1, matrix2 به همراه سطر و ستون آنها را میگیرد و میانگین نظیر به نظیر عناصر آنها را به صورت یک آرایه دوبعدی خروجی میدهد.

```
// concatenates matrices mat1 and mat2 with shape: (row,
column)
// output matrix has shape: (row, 2*column)
double** matrix_concat_2(double** mat1, double** mat2, int
row, int column);
```

تابع matrix_concat_2 در ورودی آرایه های دوبعدی mat1, mat2 به همراه سطر و ستون آنها را میگیرد و آنها را با یکدیگر کانکت کرده و خروجی میدهد.

```
// subtracts two matrices with shape (row, column)
double** matrix_subtract_2(double** matrix1, double** matrix2,
int row, int column);
```

تابع matrix_subtract_2 در ورودی آرایه های دوبعدی matrix1, matrix2 به همراه سطر و ستون آنها را میگیرد و حاصل تفریق نظیر به نظیر آنها به صورت یک آرایه دوبعدی خروجی میدهد.

```
// creates a matrix with shape (row, column) and assigns all
elements to one
double** matrix_one(int row, int column);
```

تابع matrix_one در ورودی دو عدد صحیح row, column ورودی میگیرد و یک ماتریس با این ابعاد که مقدار همه عناصر آن برابر 1 است برمیگرداند.

- حال به توضیح توابع فایل lstm_funcs.h میپردازیم.

```
// forward function of LSTM
// computes new values for h_t and c_t
double** forward_lstm(double x[batch_size][sequence_size][input_size],
double** W_i, double** U_i, double** matrix_bi, double** W_f, double**
U_f, double** matrix_bf, double** W_c, double** U_c, double** matrix_bc,
double** W_o, double** U_o, double** matrix_bo, double** c_t_param,
double** h_t_param, double initial_weight);
```

تابع forward_lstm اصلی ترین تابع بخش LSTM است.
این تابع در ورودی آرایه سه بعدی x (دنباله ورودی LSTM حاوی اعداد اعشاری) را به همراه آرایه های دوبعدی W_i, U_i, W_f, U_f, W_c, U_c, W_o, U_o, matrix_bo, c_t_param, h_t_param و عدد اعشاری initial weight را ورودی میگیرد.
قسمت اصلی این تابع قطعه کد زیر است:

```
    // iterate on each sequence and compute h_t and c_t
    for (j = 0; j < sequence_size; j++){
        for (i = 0; i < batch_size; i++){
            for (k = 0; k < input_size; k++){
                x_t[i][k] = x[i][j][k];
            }
        }

        // compute i_t
        mult_xt_Wi = matrix_mult(x_t, batch_size, input_size, W_i,
input_size,hidden_size);
        mult_ht_Ui = matrix_mult(h_t, batch_size, hidden_size, U_i,
hidden_size, hidden_size);
        sum_i = matrix_sum_3(mult_xt_Wi, mult_ht_Ui, matrix_bi,
batch_size, hidden_size);
        i_t = matrix_sigmoid(sum_i, batch_size, hidden_size);

        // compute f_t
        mult_xt_Wf = matrix_mult(x_t, batch_size, input_size, W_f,
input_size, hidden_size);
        mult_ht_Uf = matrix_mult(h_t, batch_size, hidden_size, U_f,
hidden_size, hidden_size);
```

```
        sum_f = matrix_sum_3(mult_xt_Wf, mult_ht_Uf, matrix_bf,
batch_size, hidden_size);
        f_t = matrix_sigmoid(sum_f, batch_size, hidden_size);

        // compute g_t
        mult_xt_Wc = matrix_mult(x_t, batch_size, input_size, W_c,
input_size, hidden_size);
        mult_ht_Uc = matrix_mult(h_t, batch_size, hidden_size, U_c,
hidden_size, hidden_size);
        sum_c = matrix_sum_3(mult_xt_Wc, mult_ht_Uc, matrix_bc,
batch_size, hidden_size);
        g_t = matrix_tanh(sum_c, batch_size, hidden_size);

        // compute o_t
        mult_xt_Wo = matrix_mult(x_t, batch_size, input_size, W_o,
input_size, hidden_size);
        mult_ht_Uo = matrix_mult(h_t, batch_size, hidden_size, U_o,
hidden_size, hidden_size);
        sum_o = matrix_sum_3(mult_xt_Wo, mult_ht_Uo, matrix_bo,
batch_size, hidden_size);
        o_t = matrix_sigmoid(sum_o, batch_size, hidden_size);

        // compute c_t
        mult_ft_ct = matrix_product(f_t, c_t, batch_size, hidden_size);
        mult_it_gt = matrix_product(i_t, g_t, batch_size, hidden_size);
        c_t = matrix_sum_2(mult_ft_ct, mult_it_gt, batch_size,
hidden_size);

        // compute h_t
        tanh_ct = matrix_tanh(c_t, batch_size, hidden_size);
        h_t = matrix_product(o_t, tanh_ct, batch_size, hidden_size);
    }

    return h_t;
```

توضیح مختصر این قطعه کد به این صورت است که سه حلقه for تو در تو روی دنباله ورودی زده میشود و هر بار یک آرایه دوبعدی x_t از دنباله ورودی استخراج میشود.
در هر بار اجرای حلقه، محاسبات طبق فرمول های زیر انجام میشود:

$$f_t = \sigma(W_f\, x_t + U_f\, h_{t-1} + b_f)$$

$$i_t = \sigma(W_i\, x_t + U_i\, h_{t-1} + b_i)$$

$$o_t = \sigma(W_o\, x_t + U_o\, h_{t-1} + b_o)$$

$$g_t = \tanh\ (W_g\, x_t + U_g\, h_{t-1} + b_g) \text{ a.k.a. } \tilde{c}_t$$

$$c_t = f_t \circ c_{t-1} + i_t \circ g_t$$

$$h_t = o_t \circ \tanh\ (c_t)$$

در نهایت پس از پایان اجرای حلقه ها، مقدار نهایی h_t که همان hidden state است خروجی داده میشود.

```
// forward function of LSTM
// computes new values for h_t and c_t
double** backward_lstm(double x[batch_size][sequence_size][input_size],
double** W_i, double** U_i, double** matrix_bi, double** W_f, double**
U_f, double** matrix_bf, double** W_c, double** U_c, double** matrix_bc,
double** W_o, double** U_o, double** matrix_bo, double** c_t_param,
double** h_t_param, double initial_weight);
```

تابع backward_lstm شبیه به تابع forward_lstm میباشد.
تنها تفاوت این دو تابع این است که در تابع backward_lstm، دنباله ورودی از انتها به ابتدا پیمایش میشود.
پس از انجام محاسبات و پایان اجرای حلقه ها، مقدار نهایی h_t خروجی داده میشود.

```
double** vectorized_forward_lstm(double
x[batch_size][sequence_size][input_size],
double** W, double** U, double** matrix_bias, double** c_t_param,
double** h_t_param, double initial_weight);
```

تابع vectorized_forward_lstm، همان کار تابع forward_lstm را انجام میدهد اما فرمول های آن کمتر است و همان طور که از اسم آن پیداست، عمل vectorization روی فرمول های قبلی اعمال شده که باعث میشود محاسبات سریعتر انجام شوند.
قسمت اصلی این تابع قطعه کد زیر است:

```
    // iterate on each sequence and compute h_t and c_t
    for (j = 0; j < sequence_size; j++){
        for (i = 0; i < batch_size; i++){
            for (k = 0; k < input_size; k++){
                x_t[i][k] = x[i][j][k];
            }
        }

        // compute gates
        mult_xt_W = matrix_mult(x_t, batch_size, input_size, W,
input_size, 4*hidden_size);
        mult_ht_U = matrix_mult(h_t, batch_size, hidden_size, U,
hidden_size, 4*hidden_size);
        gates = matrix_sum_3(mult_xt_W, mult_ht_U, matrix_bias,
batch_size, 4*hidden_size);

        // compute i_t
        for (m = 0; m < batch_size; m++){
            for (n = 0; n < hidden_size; n++){
                sliced_gates_i[m][n] = gates[m][n];
            }
        }

        i_t = matrix_sigmoid(sliced_gates_i, batch_size, hidden_size);


        // compute f_t
        for (m = 0; m < batch_size; m++){
            for (n = 0; n < hidden_size; n++){
                sliced_gates_f[m][n] = gates[m][n];
            }
        }

        f_t = matrix_sigmoid(sliced_gates_f, batch_size, hidden_size);


        // compute g_t
        for (m = 0; m < batch_size; m++){
            for (n = 0; n < hidden_size; n++){
                sliced_gates_g[m][n] = gates[m][n];
```

```
            }
        }

        g_t = matrix_tanh(sliced_gates_g, batch_size, hidden_size);


        // compute o_t
        for (m = 0; m < batch_size; m++){
            for (n = 0; n < hidden_size; n++){
                sliced_gates_o[m][n] = gates[m][n];
            }
        }

        o_t = matrix_sigmoid(sliced_gates_o, batch_size, hidden_size);


        // compute c_t
        mult_ft_ct = matrix_product(f_t, c_t, batch_size, hidden_size);
        mult_it_gt = matrix_product(i_t, g_t, batch_size, hidden_size);
        c_t = matrix_sum_2(mult_ft_ct, mult_it_gt, batch_size,
hidden_size);

        // compute h_t
        tanh_ct = matrix_tanh(c_t, batch_size, hidden_size);
        h_t = matrix_product(o_t, tanh_ct, batch_size, hidden_size);
    }

    return h_t;
```

مشابه با تابع forward_lstm، اینجا نیز سه حلقه for تو در تو روی دنباله ورودی زده میشود و هر بار یک آرایه دوبعدی x_t از دنباله ورودی استخراج میشود.
در هر بار اجرای حلقه، محاسبات طبق فرمول های vectorized شده انجام میشود و پس از پایان اجرای حلقه ها، مقدار نهایی h_t خروجی داده میشود.

```
double** vectorized_backward_lstm(double
x[batch_size][sequence_size][input_size],
double** W, double** U, double** matrix_bias, double** c_t_param,
double** h_t_param, double initial_weight);
```

تابع vectorized_forward_lstm، مشابه با تابع vectorized_forward_lstm است.
تنها تفاوت این دو تابع این است که در تابع vectorized_forward_lstm، دنباله ورودی از انتها به ابتدا پیمایش میشود.

```
double** bi_lstm(double** h_t_forward, double** h_t_backward,
char merge_mode[]);
```

آخرین تابع فایل lstm_funcs.h، تابع bi_lstm است.
این تابع آرایه های دوبعدی h_t_forward, h_t_backward و رشته ی merge_mode را در ورودی میگیرد.
این دو آرایه، خروجی توابع forward_lstm , backward_lstm (یا نسخه vectorized آنها) هستند و رشته merge_mode، حالت ترکیب شدن آنها را مشخص میکند.
مشابه با پیاده سازی لایه lstm در keras، حالت ترکیب میتواند یکی از حالت های "concat", "avg", "mult, "sum" باشد.
- در نهایت در فایل lstm_main.c ماتریس های وزن مقدار دهی اولیه میشوند و توابع backward_lstm، forward_lstm و bi_lstm صدا زده شده و خروجی آنها چاپ میشود.
خروجی سه تابع در کد c و کد python به ازای پارامتر ها و ورودی های یکسان (عکس اول خروجی کد c، عکس دوم خروجی کد python):

```
**********************************************************

forward lstm output:

0.849751        0.849751
0.907997        0.907997

**********************************************************

backward lstm output:

0.839873        0.839873
0.900360        0.900360

**********************************************************

bi lstm output:

0.849751        0.849751        0.839873        0.839873
0.907997        0.907997        0.900360        0.900360

**********************************************************
```

```
**********

forward lstm output:

tensor([[0.8498, 0.8498],
        [0.9080, 0.9080]], grad_fn=<MulBackward0>)

**********

backward lstm output:

tensor([[0.8399, 0.8399],
        [0.9004, 0.9004]], grad_fn=<MulBackward0>)

**********

bi lstm output:

tensor([[0.8498, 0.8498, 0.8399, 0.8399],
        [0.9080, 0.9080, 0.9004, 0.9004]], grad_fn=<CatBackward0>)

**********
```

3.5. مطالعه مدل های Bidirectional RNN و پیاده سازی مدل Bi - LSTM به زبان c و تست آن

پس از تکمیل پیاده سازی مدل LSTM، تبدیل آن به مدل Bi - LSTM آغاز شد.
پس از مطالعه مدل های Bidirectional RNN و فهمیدن روند آنها، پیاده سازی مدل Bi - LSTM آغاز شد.
در این مرحله نیاز بود که مدل LSTM پیاده سازی شده در مرحله قبل که در واقع یک تابع بود، تغییراتی داشته باشد.
از جمله اینکه به جای اینکه تابع خروجی نداشته باشد و صرفا مقادیر یک آرایه چند بعدی global را تغییر دهد، تابع یک آرایه چند بعدی (h_t) را خروجی دهد که بتوان خروجی آن را در تابع Bi - LSTM استفاده کرد.
پس از انجام تغییرات لازم، تابع دیگری به نام backward lstm پیاده سازی شد.
لازم به ذکر است که تابع قبلی پیاده سازی شده، forward lstm نام داشت.
تابع backward lstm، مشابه تابع قبلی است اما صرفا ورودی ها رو به صورت برعکس پیمایش میکند و آرایه h_t را خروجی میدهد.
در نهایت تابع Bi - LSTM پیاده سازی شد که دو آرایه خروجی توابع قبلی و نوع ترکیب آنها را نیز دریافت میکند و با توجه به نوع ترکیب مشخص شده، آنها را با یکدیگر ترکیب می کند و خروجی میدهد.
تابع Bi - LSTM نیز مشابه با تابع LSTM، تست شد و خروجی های یکسان از کد های c, python گرفته شد.
در نتیجه صحت پیاده سازی این تابع نیز تایید شد.
پس از اتمام بخش Bi - LSTM، بخش GRU آغاز شد.

3.6. مطالعه مدل GRU

در این مرحله به مطالعه در مورد ساز و کار و نحوه کار کردن مدل GRU پرداخته شد.

پس از فهمیدن مفهوم کلی این مدل، پیاده سازی مدل آغاز شد.

3.7. پیاده سازی مدل GRU به زبان c و تست آن

در این مرحله به دلیل شباهت مدل های GRU و LSTM، کار سبک تری نسبت به شروع پیاده سازی مدل LSTM وجود داشت.
فرمول های مدل GRU در این مرحله پیاده سازی شدند.
همچنین توابعی که برای پیاده سازی این فرمول ها نیاز به پیاده سازی داشتند، پیاده سازی شدند.
پس از اتمام پیاده سازی، تست صحت پیاده سازی انجام شده مشابه با بخش LSTM انجام شد.
خروجی کد های c, python یکسان بودند.
در نتیجه صحت پیاده سازی تابع GRU تایید شد.
سپس باید مدل Bi - GRU پیاده سازی میشد.

3.8. پیاده سازی مدل Bi - GRU به زبان c و تست آن

این مرحله مشابه با پیاده سازی مدل Bi - LSTM انجام شد.
یعنی در ابتدا یک تابع دیگر به نام backward GRU مشابه با تابع forward GRU پیاده سازی شد.
سپس تابع Bi -GRU پیاده سازی شد که آرایه های خروجی توابع forward , backward و نوع ترکیب آنها را ورودی میگیرد و آرایه ترکیب شده را خروجی میدهد.
تست این تابع مشابه با تابع Bi - LSTM انجام شد و خروجی یکسان از کد های c , python گرفته شد.
در نتیجه صحت پیاده سازی این تابع نیز تایید شد.
پس از اتمام پیاده سازی های مدل های Bi - GRU , Bi - LSTM، سنتز کد های c و تبدیل به کد HDL شروع شد.

3.9. توضیح توابع پیاده سازی شده برای GRU

در این قسمت توابع پیاده سازی شده به زبان c توضیح داده میشوند.
- ابتدا توابع فایل gru_funcs.h توضیح داده میشوند.

```
double** forward_gru(double
x[batch_size][sequence_size][input_size], double** W_z,
double** U_z, double** matrix_bz, double** W_r, double** U_r,
double** matrix_br, double** W_h, double** U_h,
double** matrix_bh, double** h_t_param, double
initial_weight);
```

تابع forward_gru اصلی ترین تابع بخش GRU است.
این تابع در ورودی آرایه سه بعدی x (دنباله ورودی LSTM حاوی اعداد اعشاری) را به همراه آرایه های دوبعدی W_z, U_z, matrix_bz, W_r, U_r, matrix_br, W_h, U_h, matrix_bh, h_t_param و عدد اعشاری initial weight را میگیرد.
قسمت اصلی این تابع قطعه کد زیر است:

```
    // iterate on each sequence and compute h_t and c_t
    for (j = 0; j < sequence_size; j++){
        for (i = 0; i < batch_size; i++){
            for (k = 0; k < input_size; k++){
                x_t[i][k] = x[i][j][k];
            }
        }

        // compute z_t (update gate)
        mult_xt_Wz = matrix_mult(x_t, batch_size, input_size, W_z,
input_size, hidden_size);
        mult_ht_Uz = matrix_mult(h_t, batch_size, hidden_size, U_z,
hidden_size, hidden_size);
        sum_z = matrix_sum_3(mult_xt_Wz, mult_ht_Uz, matrix_bz,
batch_size, hidden_size);
        z_t = matrix_sigmoid(sum_z, batch_size, hidden_size);

        // compute r_t (reset gate)
        mult_xt_Wr = matrix_mult(x_t, batch_size, input_size, W_r,
input_size, hidden_size);
        mult_ht_Ur = matrix_mult(h_t, batch_size, hidden_size, U_r,
hidden_size, hidden_size);
        sum_r = matrix_sum_3(mult_xt_Wr, mult_ht_Ur, matrix_br,
batch_size, hidden_size);
        r_t = matrix_sigmoid(sum_r, batch_size, hidden_size);

        // compute h_tilde (intermediate memory)
        mult_xt_Wh = matrix_mult(x_t, batch_size, input_size, W_h,
input_size, hidden_size);
        prod_r_ht = matrix_product(r_t, h_t, batch_size, hidden_size);
        mult_prod_r_ht_and_Uh = matrix_mult(prod_r_ht, batch_size,
hidden_size, U_h, hidden_size, hidden_size);
        sum_h_tilde = matrix_sum_3(mult_xt_Wh, mult_prod_r_ht_and_Uh,
matrix_bh, batch_size, hidden_size);
        h_tilde = matrix_tanh(sum_h_tilde, batch_size, hidden_size);

        // compute h_t
        prod_z_ht = matrix_product(z_t, h_t, batch_size, hidden_size);
```

```
        one_minus_zt = matrix_subtract_2(ones, z_t, batch_size,
hidden_size);
        prod_one_minus_zt_and_h_tilde = matrix_product(one_minus_zt,
h_tilde, batch_size, hidden_size);
        h_t = matrix_sum_2(prod_z_ht, prod_one_minus_zt_and_h_tilde,
batch_size, hidden_size);
    }

    return h_t;
```

توضیح مختصر این قطعه کد به این صورت است که سه حلقه for تو در تو روی دنباله ورودی زده میشود و هر بار یک آرایه دوبعدی x_t از دنباله ورودی استخراج میشود.
در هر بار اجرای حلقه، محاسبات طبق فرمول های زیر انجام میشود:

$$z = \sigma(W_z \cdot x_t + U_z \cdot h_{(t-1)} + b_z)$$
$$r = \sigma(W_r \cdot x_t + U_r \cdot h_{(t-1)} + b_r)$$
$$\tilde{h} = tanh(W_h \cdot x_t + r * U_h \cdot h_{(t-1)} + b_z)$$
$$h = z * h_{(t-1)} + (1 - z) * \tilde{h}$$

در نهایت پس از پایان اجرای حلقه ها، مقدار نهایی h_t که همان hidden state است خروجی داده میشود.

```
double** backward_gru(double x[batch_size][sequence_size][input_size],
double** W_z, double** U_z, double** matrix_bz, double** W_r, double**
U_r, double** matrix_br, double** W_h, double** U_h, double** matrix_bh,
double** h_t_param, double initial_weight);
```

تابع backward_gru شبیه به تابع forward_gru میباشد.
تنها تفاوت این دو تابع این است که در تابع backward_gru، دنباله ورودی از انتها به ابتدا پیمایش میشود.
پس از انجام محاسبات و پایان اجرای حلقه ها، مقدار نهایی h_t خروجی داده میشود.

```
double** bi_gru(double** h_t_forward, double** h_t_backward,
char merge_mode[]);
```

آخرین تابع فایل gru_funcs.h، تابع bi_gru است.

مشابه با تابع bi_lstm در قسمت LSTM، این تابع آرایه های دوبعدی h_t_forward, h_t_backward و رشته ی merge_mode را در ورودی میگیرد.
این دو آرایه، خروجی توابع forward_gru , backward_gru هستند و رشته merge_mode، حالت ترکیب شدن آنها را مشخص میکند.
حالت ترکیب به طور مشابه میتواند یکی از حالت های "concat" ,"avg", "mult, "sum" باشد.
- در نهایت در فایل gru_main.c ماتریس های وزن مقدار دهی اولیه میشوند و توابع backward_gru ، forward_gru و bi_lstm صدا زده شده و خروجی آنها چاپ میشود.
خروجی سه تابع در کد c و کد python به ازای پارامتر ها و ورودی های یکسان (عکس اول خروجی کد c، عکس دوم خروجی کد python):

```
**********************************************************

forward gru output:

0.261206        0.261206
0.177363        0.177363

**********************************************************

backward gru output:

0.261903        0.261903
0.177378        0.177378

**********************************************************

bi gru output:

0.261206        0.261206        0.261903        0.261903
0.177363        0.177363        0.177378        0.177378

**********************************************************
```

```
**********************************************************************

forward gru output:

tensor([[0.2602, 0.2602],
        [0.1772, 0.1772]], grad_fn=<AddBackward0>)

**********************************************************************

backward gru output:

tensor([[0.2600, 0.2600],
        [0.1771, 0.1771]], grad_fn=<AddBackward0>)

**********************************************************************

bi gru output:

tensor([[0.2602, 0.2602, 0.2600, 0.2600],
        [0.1772, 0.1772, 0.1771, 0.1771]], grad_fn=<CatBackward0>)

**********************************************************************
```

3.10. سنتز کد های c به وسیله ابزار Vivado HLS

در مرحله پایانی، کد های c با استفاده از ابزار Vivado HLS سنتز شدند و کدهای verilog, VHDL, SystemC معادل آنها خروجی گرفته شدند.

4. خلاصه

به طور کلی هدف از این کارآموزی پیاده سازی سخت افزاری و سطح پایین مدل های Bi - LSTM , Bi - GRU بود.
فرآیندی که برای رسیدن به این هدف طی شد شامل تحقیق و مطالعه مدل های LSTM , GRU، یافتن پیاده سازی مناسب python و تبدیل آن پیاده سازی به زبان c و سنتز کد c و تبدیل آن به کد سخت افزاری بود.
مهمترین چالش های این فرآیند، درک مفهوم این مدل ها و محدودیت های زبان c برای پیاده سازی آنها بودند.

نظرات کارآموز جهت بهبود کارآموزی

به کارآموزان بابت کار انجام شده حقوق پرداخت شود.

ضمایم