Linux Kernel Code - On Demand

– Text Prediction Using Recurrent Neural Networks –

Martin Estgren `<mares480@student.liu.se>`

Linköping University, 19th January 2018

Abstract

The Linux Kernel project suffers from a significant problem, thousands of people are collaborating to create one of the most popular contemporary Operating System Kernels. There are fears that software project of such magnitude may cause bugs which no single human is able to solve[1]. In this paper we will examine how, by training a *Recurrent Neural Network* to generate new kernel code, the productivity and efficiency of the Linux Project might increase as human involvement can be limited to oversight and managerial tasks.

Contents

1 Introduction

The use of *Recurrent Neural Networks* (RNN) for text prediction and classification has seen great results in the last couple of years [4]. A. Karpathy presents in his excellent blog post how RNN models can be used to approximate close to arbitrary character sequences, such as Shakespeare, LaTeX, or C source code. In this project we will aim to produce procedurally generated source code for the Linux kernel using a RNN model.

The paper starts with a section about important theoretical background to the method. After the theory, implementation details such as RNN model and *hyper parameters* will be presented followed by results and finally a discussion and conclusion about the project findings.

[1]`https://lwn.net/Articles/285088/`

2 Theory

This section presents important aspects of relevant model design such as network architecture and evaluation function. For a more in-depth reading, see the textbook written by *I. Goodfellow et. al* [1].

2.1 Recurrent Neural Network (RNN)

As described by *I. Goodfellow et. al* [1], *RNN*s are a family of deep learning models which specializes in predicting outputs dependent on a sequential input. In general terms can a RNN be seen as the unrolling of an regular *Feed forward Neural Network* where the output is connected to the input of a part of the network. Such feedback loops allows the network to model an internal state. The figure 1 shows a typical *Recurrent Neural Network* where there is a feedback loop between parts of the *hidden layers*.

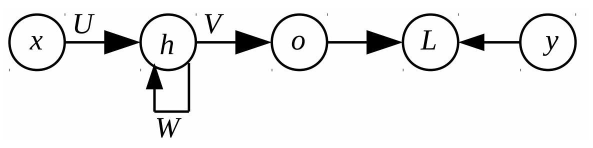

Figure 1: Example of an RNN with a feedback loop in the hidden layers. x is the input and L a function which measures the *error* from predicted output (o) and the expected value y. W, V , and U are matrices with weights for each of the layers.

Unrolling the network (as shown in figure 2) makes it easier to observe how the looping structure allow for sequence dependent output with shared weights between each time step. This is a significant advantage in terms of model complexity. For example, a input in the time-step $t-5$ could make the output in t fire.

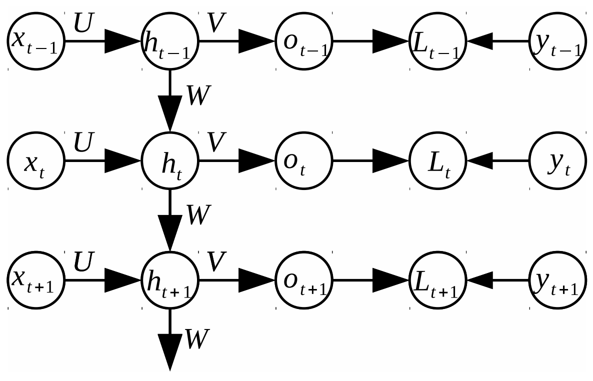

Figure 2: Unrolling of a neural net (N) with feedback loops. t represents the given *time-step* and h the output for the given *time-step*.

In theory, a *RNN* model, as shown above in figure1, can model functions where the output depends on input which has arrived a long time before the current time-step. However, according to [3], the most widely used methods requires impractically much time to model such long-term memory models. The main reason for this discrepancy is the problem of *vanishing gradients*, where the propagation error significantly reduces as layers further from the output are corrected.

2.2 Long Sort Term Memory (LSTM)

Long Sort Term Memory Networks was first proposed by *S. Hochreiter and J. Schmidhube* [3] as a remedy to the problem of modelling long-term memory in *RNN*s. The paper propose an extension to the typical hidden RNN cell, which allow for easier long-term memory as the *back-propagation* does not suffer as much by *vanishing gradients*.

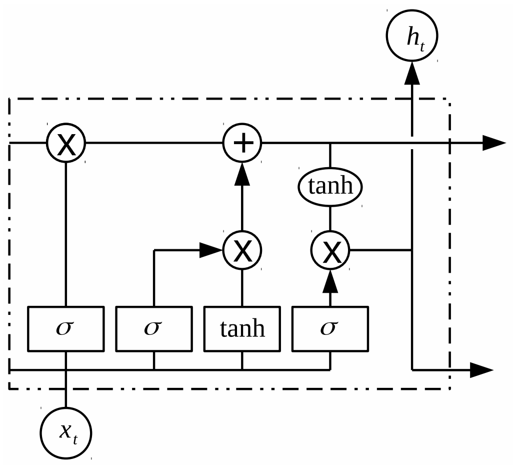

Figure 3: Visualization of an *LSTM* cell. Each block is a Layer operation and the circles are elementwise. Credit to C. Olah for the excellent blog post[5] dissecting the function of LSTM cells.

Figure 4 shows the same *LSTM* cell but with internal variables annotated.

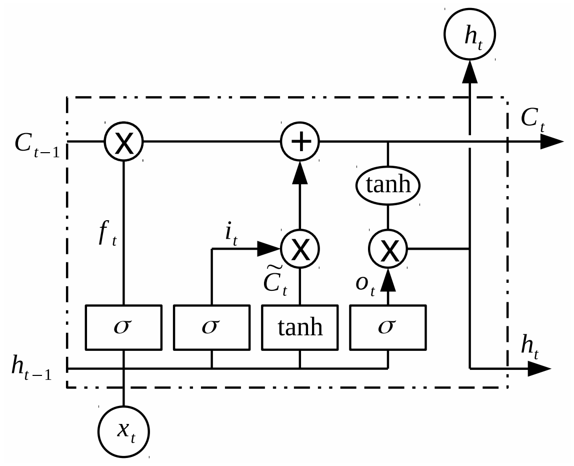

Figure 4: Annotated version of a *LSTM* cell.

There are four layers in each cell, three σ layers and one *tanh*. The fist sigmoid layer is referred to as a *forget gate*, which determines what parts of the signal from the previous time-steps should continue propagation. The output of the gate is determine on the form:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t]) \tag{1}$$

where W_f is the weights for said layer (intercept values are assumed to be included in the weight-matrix for all layer operations).

The second σ and the *thanh* layers are combined in order to determine what should be added to the memory line. It is computed using the equations:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t]) \tag{2}$$

$$\tilde{C}_t = tanh(W_C \cdot [h_{t-1}, x_t]) \tag{3}$$

which are combined on the form $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$. The output for the cell t is computed using the memory-line and the last σ layer:

$$o_t = \sigma(W_o[h_{t-1}, x_t]) \tag{4}$$

$$h_t = o_t * tanh(C_t) \tag{5}$$

where h_t is the final output from the cell.

2.3 Loss function

The loss function used to compare true and predicted character sequences is a variant of *cross entropy* for multinomial classes based on the *softmax* function. Credit to *P. Roelants*[2] for the derivation of loss function. In principe we take the *softmax* function:

$$y_c = \frac{e^{z_c}}{\sum_{d=1}^{C} e^{z_d}} \tag{6}$$

where C is the set of all classes we want to evaluate. Applying the function for all classes yelds a vector with values $\in [0, 1]$ and element sum of 1.

Calculating the *cross-entropy* loss function for the *softmax* function is done though:

$$\xi(T, Y) = -\sum_{i=1}^{n} \sum_{t=c}^{C} t_{ic} \cdot log(y_{ic}) \tag{7}$$

where T is a vector of all expected class labels and Y is a vector with the corresponding predicted labels. The outer sum allows to perform the loss calculation for multiple characters, for example a batch or a sequence.

3 Implementation Details

The model will be evaluated on a computer with the following hardware:

RAM	16GB
CPU	AMD Ryzen 7 170
GPU	NVIDIA GTX 980Ti 6GB

Table 1: Hardware specification for computer which will be used for model training and predictions.

The model will be implemented in the *Tensorflow* framework and additional code is implemented in *Python*.

3.1 Data gathering and Representation

Character sequences used for the model training are extracted from the Linux kernel source code repository [3]. All files with the ending ".c" are concatenated and the first 1 million lines are used as training/validation data.

An alphabet of all occurring characters is created and used for encoding/decoding the model input

[2]http://peterroelants.github.io/posts/neural_network_implementation_intermezzo02/

[3]https://github.com/torvalds/linux

and output respectively. The alphabet used in order to model the network contains the following characters:

```
[
'\t', '\n', '\x0c', ' ', '!', '"',
'#', '$','%', '&', "'", '(', ')',
'*', '+', ',', '-','.', '/', '0',
'1', '2', '3', '4', '5', '6','7',
'8', '9', ':', ';', '<', '=', '>',
'?','@', 'A', 'B', 'C', 'D', 'E',
'F', 'G', 'H','I', 'J', 'K', 'L',
'M', 'N', 'O', 'P', 'Q','R', 'S',
'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
'[', '\\', ']', '^', '_', '`', 'a',
'b', 'c','d', 'e', 'f', 'g', 'h',
'i', 'j', 'k', 'l','m', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u','v',
'w', 'x', 'y', 'z', '{', '|', '}', '~'
]
```

which results in the *one-hot* input vector $\vec{x} \in \mathbb{Z}_2^{98}$. The output vector $\vec{y}$ from the model will be on the form $\vec{x} \in \mathbb{R}^{98}$, where each row corresponds to a symbol in to alphabet and the probability of said symbol appearing as the next character in the sequence.

3.2 Network Architecture

The network architecture is built with one input layer, 4 hidden layers, and a fully connected output layer. The hidden layers consists each of 256 LSTM cells. As described in the section above, the input and output are both vectors with 98 dimension. Layer weights are randomly initiated around 0 with a standard deviation of 0.01. The output layer uses a *softmax* function to predict the probability of the next character. During the testing phase, these probabilities are used to pick the next character and the result is once again feed into the network.

The significant hyper parameters are listed below:

- **Hidden Layer Size** (256) - Number of *LSTM* nodes in each of the hidden layers.
- **Number Hidden Layers** (4) - Number of hidden layers.
- **Batch Size** (64) - Number of sequences for each training iteration.
- **Time steps** (256) - Length of the character sequence used during training.
- **Loss function** (softmax with binary cross entropy) - Function used to determine the weight adjustments for the back-propagation pass.

During the testing phase, character sequences of length 2000 are generated with the seed "#include "

4 Results

The figure below shows the loss function during the training phase. We can observe how the error converges around 0.7. In total it took 9.53 hours to train the model with a limit off 50000 batches.

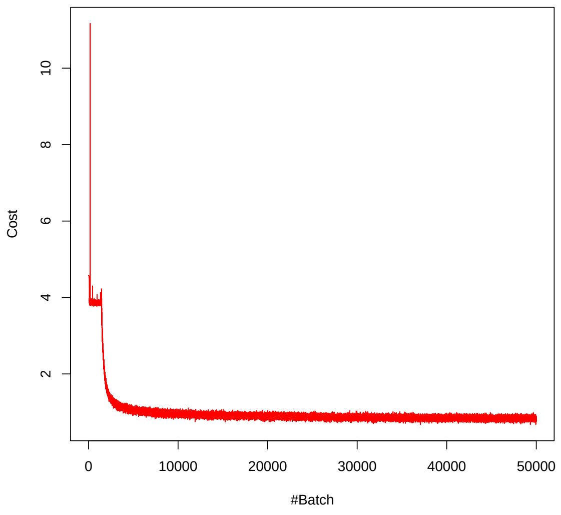

Figure 5: Error value over batch number. The x-axis corresponds to training iterations and the y-axis, the loss. Lower loss is better.

Below are sequences generated with the model. For the sake of readability is a full output sequence included as *Appendix A*.

```
#include <linux/vector.h>
#include <linux/fs.h>
#include <linux/rstab_lock.h>
#include <linux/platform_image.h>
#include <linux/pci.h>
#include <linux/topolution.h>
#include <linux/mm.h>
#include <linux/init.h>
#include <linux/init.h>
#include <linux/irq.h>
#include <linux/init.h>
#include <linux/device.h>
#include <linux/platform_data/usb-outkhake.h>
```

The listing above show how the model have predicted a set of include definitions. Notice how many includes are repeated but correctly formatted.

```
/**
 * enable_single_stepdatormance_regs()) for
 * @sg:  "problem" the access a pointer of reset has a rounding.
 * @kunotbgeWalreaschdr_info: The record for MMIO PLL core written by the id
 * @regs: Process pointary
 *
 * Copyright (C) 2010, Gregen Andrs Culd <sholk@deak2.c
 */
```

The above listing shows a multi line comment with a copyright notice included. Each line starts with a "*" which is not required by the C definition but a common practice.

The content following output is a function declaration with a repeated parameter list, void return type and bitwise operations. We can also observe pointer arithmetic and constants written in UPPERCASE.

```
void kvm_err_exception_task(struct kvm_vcpu *vcpu,
                                          struct kvm_vcpu *vcpu)
{
        struct kvmppc_get_krc_page (bfin_read12s, &timer_emulator);

        pmces = regs->uregs[power] & (HPTE_V_H | LOCOCON_FLUST | COMPAT_HWCAP_B) |
                                   pfm_states->controllers_mask[i];
}
```

5 Discussion and Conclusion

Looking at the source code generated by the model, we can observe how it generalizes some trends in the C syntax, such as opening and closing block commends. It does not however keep track of variables in scope, resulting in variable names which are not declared. Even though it is clear that the model generalized some aspects of the source code. It cannot generate anything more coherent than isolated clauses. Given that the input was a combination of many contextually different parts of the source code, the problem of keeping variables in scope might be explained.

Below is a listing where a excerpt of the training data is shown. The model generate sequences which are incredibly similar on a visual basis:

```c
/**
 * smp_show_cpu_info - Show SMP CPU
   information
 * @cpu: The CPU of interest.
 */
static void __init smp_show_cpu_info(int
    cpu)
{
    struct mn10300_cpuinfo *ci = &cpu_data
    [cpu];

    printk(KERN_INFO
           "CPU#%d : ioclk speed: %lu.%02
    luMHz : bogomips : %lu.%02lu\n",
           cpu,
           MN10300_IOCLK / 1000000,
           (MN10300_IOCLK / 10000) % 100,
           ci->loops_per_jiffy / (500000 /
     HZ),
           (ci->loops_per_jiffy / (5000 /
    HZ)) % 100);
}
```

Surprisingly, the model generalized the fact that within comments, the character sequences are closer to human languages instead of source code. This would be interesting if one were to create a classification model instead, differentiating between natural and programming languages.

It is also possible that the *by-character* approach to sequence generation might be to primitive for the net size and time constraints of the project. A better approach might be to group characters into semantically relevant words. This would result in a much larger input and output vector but would put less pressure on the model to both generalize both the language grammar and semantics.

The size of the model was limited by time and computational constraints on the project. Better hardware may allow for larger models which performs better.

In conclusion we can say that although the result looks promising at a cursory glance, the technique is far from able to produce valid C source code. Previous studies have shown that RNN models using LSTM are incredibly powerful[2] (in fact, Turing complete [6]) but suffers from the same problem as other deep learning models, computational time and access to labelled data. Perhaps in one or tow AI winters the technique is closer to compete with humanoid software developers.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[2] K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*, 2015.

[3] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[4] A. Karpathy. The unreasonable effectiveness of recurrent neural networks. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`. Accessed: 2018-01-16.

[5] C. Olah. Understanding lstm networks. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`. Accessed: 2018-01-17.

[6] H. T. Siegelmann. Computation beyond the turing limit. *Science*, 268(5210):545–548, 1995.

A Predicted Model Output

```
#include <linux/suspend.h>
#include <linux/intspec.h>
#include <linux/types.h>
#include <linux/input/master.h>
#include <linux/mtd/physmap.h>
#include <linux/platform_data/usb-ohci-pxa.h>
#include <linux/platform_data/irda-pxaficp.h>
#include <linux/platform_data/usb-ohs/n4";
elak_def:
        switch (flubly) {
        wakeupgen_worksz[0] = AR2315_PWRCAML_GE_CONF ^ EMWEL;

        /* restoring the soc timer is written. */
        struct omap_dm_timer *ops = &s3c_i2c3_irq_mask, &ids[1];
        phy_device_config.s.mrite = 0;
#ifdef CONFIG_SOC_CTS intcsp(void)
        unsigned char buf[138];
        omap_prcm_init();
        if (!data) {
                pr_err("FAIL: Code == marked states\n");
                mv74xx_cb58080 = 1;
                machine_has_cpumask_init();

                switch (reg) {
                       __asm__ volatile("mrc != "n2"); \
                        { MCA_IAR4--, }, \
                        /*
                 * Permatchs. This kernel with the dma_1
                   without interpreter-signals" max/diggeciun/bits called.
         *              xQlist on the intended
                   boot/timer in the temporary process TLB and we posts
                   warn-at percpu to a fec/mask.  Performs over this and sigbus
         * shift. Enable works mode:
         */
        if (writeb(data & (PAGE_SHIFT) >>
           (H_PAGE_BUSX)) {
                const int yev;
        }

        if (count < (unsigned int)-1)   {
                return -EPERM;
        }

        /* preserve variation is only early in hardware */
        *val = 0;

        /* Cliculition state to be only only courity */
        if (!value) {
                ret = -ENOENT;
                goto pc_mbus;
        }
        pgprot_noncached(vaddr); /* unless the point it setting to one. */
        if (PAGE_ALIGN(addr < mm)) {
                /* and this fun the pgd type bit 0 */

                flush_altivec_to_thread(ptep);
                Bstackptep(start, pte, transto_gid, fn);
        }
        tlb_destination_t nat;
};

void kvm_err_exception_task(struct kvm_vcpu *vcpu,
                                          struct kvm_vcpu *vcpu)
{
        struct kvmppc_get_krc_page (bfin_read12s, &timer_emulator);

        pmces = regs->uregs[power] & (HPTE_V_H | LOCOCON_FLUST | COMPAT_HWCAP_B) |
                                    pfm_states->controllers_mask[i];
}

static void __init hub_rt_read_match(ptr, size_t ptr, unsigned long r5, unsigned int n_ap, unsigned char *dprom)
{
        u32 range_index;                "insn2", &dstptr->tlbsize);
        unsigned int *mask = 4;
        unsigned int long long r3, r6;
        struct rtc_type data;
#ifdef CONFIG_VOLPAIRENTEDERS
        unsigned int bounce;
        unsigned long sp; /* SCXE1>  *soctable is programmy */
        unsigned int mask, cache = 0, result = 0;

        /* Set D error taken bit */
        while (state->last_insn == 0) {
                union cvmx_pko_denormal32 attr;
                struct uasm_ral update_reg = val |
                        SYS_RESVOLG_EN_IP20 | FRV_DMA_CAP_USATEV | ARMV6_PF0_USE_PPS_OFF;
                scuproves_type = REG_TJSCR;
                cnstcr = 0;
                /* Passes to save dive bit */
                result.s.len = 0;
                res = 0;
                BUG_ON(dstk != oud);
                if (r_exponent != state.fpu_free_x5)
                        return x faul;
                BUG_ON(ctx->ram == (int)Multiplely A not set\n");
                return;
        }

        if (reg.s.mask || !9) {
                current->thread.dsp_context[EXTBASEt_exponent(src)].fSignandrate *dsend =
                        (insn.format.f) != -1 ? 'U' : '\'H'\3a, 59, 80+8);      /* dst */
                __x3 += SB_n = (MIPSInst_RT(ir): count > 0 || x != 0x7FFFFFFF); \
                prog->source = netrof_vector_typ; bot++;                \
        }                                       \
I_r32rap                \
                                                        \
        write_gc0_guest_config(str, ctx); \
        } while (0)

#define FLASH_BL        0x1c            /* #slave and R9 */
#define RECOR           ? 1
#define FPSR_F1         0xf
#define SETNAT_IO_SHARED                0x40
#define SETXER_CTLB     0x00000000      /* 0000 %016> -> QRR x12 buf *xics,bchip@asm*\n"); \
enet150 */
static int rb532_register_devices[] __initdata = {
        &s3c_device_attribute, {
                .virtual        = Paulbase,
                .collistrate            = kvmppc_pa01_ops =
    kvm_mips_dequeue_irq,
                .existat_bits   = 1,
        },
#endif

        /* .SigN machine data flash, thus priority, if not the PCI interproc'llads */
        mdelay(1);
        while (PWMCTL_PAR_PB(pd) {
                pdata = kzalloc(PCMCLK_H1, handle_backlight);
                mdelay(10);
                gdbstub_alloc(&dev->resource[0].page, 0);
                res->end  = ARM_BREAKPOINT_W;
                gfp &= ~GFC_RWSTCTRL;
        } else {
                self_pte = mmu_valid_address);
                ret = efi_map_start - tlbmiss_size;
        }
        if (!prom_argv) {
                prot = PTRS(pteg, (u32) PAGE_ALIGN(vmap),
                        size, pgprot_val(pgprons[0]));
                __halted = 1;
                _mmio_unuter_cache_tag[0].prot / bitshiftb1;
```