Математичка гимназија

МАТУРСКИ РАД
- из информатике -

Машиншко учење у Rust-у

Ученик:
Јован Марковић IVд

Ментор:
Мијодраг Ђуришић

Београд, јун 2021.

Садржај

1

Увод

Вештачка интелигенција се односи на симулацију људске интелигенције на машинама које су програмиране да мисле као људи. Машиншко учење је један од видова вештачке интелигенције који оспособљава системе да аутомацки уче и побољшавају се из искуства. Машиншко учење се фокусира на развој програма који користећи неки скуп података имају способност да се самоунапреде.

Rust је програмски језик чији је циљ ефикасност, меморијска сигурност. Rust, такође, има синтаксу која је донекле читљива свима. Ове особине имају неке интересантне импликације, једна од којих је да се разне области које се примарно раде у језицима висег нивоа, могу једнако лако радити у Rust-у. Како не постоје библиотеке за машинско учење доступне за Rust, позабавићемо се писањем једне.

2

Машиншко учење и неуронске мреже

Алгоритми за машиншко учење се могу поделити на надгелдане(supervised), ненадгледане(unsupervised) и поткрепљене(reinforcement). Надгледани алгоритми полазе од анализе познатих, обележених, података помоћу којих уче апроксимативну функцију којом ће мочи да праве предвиђања за улазе који нису коришћени у току тренирања. Ненадгледани алгоритми се користе кад информација која треба да се научи није класификована ни обележена. Они се користе како би се научила функција која описује скривене структуре међу подацима. Систем који користи ненадгледано учење истражује податке и може да извуче закључке о везама између њих. Поткрепљени алгоритам је метод учења којим систем производећи радње интерагује са динамичким окружењем и добија казне и награде. Овај метод учења омогућава машинама и софтверским агентима да аутомацки одреде идеално понашање у специфицном контексту да би максимизовали награду.

Вештачке неуронске мреже(ANN-artificial neural network), или само неуронске мреже, су структуре које су донекле моделоване по биолоским неуронским мрежама. ANN је базирана на колекцији повезаних цворова званих вештачки неурони. Неурони су међусобно повезани и сваки од њих прими информације од неких неурона, обради их и проследи другим неуронима. ANN је најчешће имплементирана тако да свака веза има тежину чија је вредност реалан број, а излаз сваког неурона је сума свих његових улаза кад се на њу примени нека активациона функција. Вредности веза се подешавају током учења. Вештачки неурони су типично организовани у слојеве. Сваки слој је само низ вештачких неурона. Неурони унутар слоја су ретко међусобно повезани. Низ слојева сачињава ANN.

2.1 Примене

Неуронске мреже се користе за:

- Апроксимације функција и регресиону анализу
- Класификацију и препознавање образаца и секвенци
- Обраду података, укључујући филтрирање, кластеровање и компресију

Због способности да репродукују и моделирају нелинеарне процесе, ANN имају примене у многим дисциплинама. Неке од тих дисциплина су квантна хемија, препознавање образаца(препознавање лица, класификација сигнала, 3Д реконструкција, препознавање објеката,...), препознавање секвенци(говор, писан и куцан текст), медицинска диагноза, финансије, data mining, машинско преводјење, ... ANN су, такође, коришћене да препознају неколико врста канцера. ANN се употребљавају и у рачунарској безбедности, за диференцирање између легитимних и злонамерних радњи. Постоје предлози да се ANN користе за решавање парцијалних диференцијалних једначина, што би омогућило симулацију неких физицких система.

2.2 Зашто неуронске мреже раде?

На основу описа неуронских мрежа не постоји интуитиван разлог зашто оне раде. Из Universal Approximation теореме се може добити да неуронске мреже са довољно параметара могу да произвољно добро апроксимирају сваку well-behaved функцију. Ова теорема излази из оквира овог рада.

2.3 Унапређења на ANN

Најједноставнија ANN је састављена од низа слојева који су састављени од неурона. У оваквој ANN је сваки неурон у једном слоју повезан са сваким у претходном слоју. Овакви слојеви се зову dense(густ) слојеви. Поред Dense слојева постоје и други попут конволутивних и pooling слојева. Ова три слоја улазе у састав конволутивних неуронских мрежа(CNN-convolutional neural network). CNN се примарно користе за обраду слика, али имају примене свуда где подаци који се обрађују имају мрежну топологију. Иако ANN могу да се истренирају да раде исти посао, постају веома непрактичне за веће улазе, јер захтевју много већи број неурона сто значи и већи број параметара који се тренирају. Осим конволутивних, pooling и dense слојева, постоје и многи други

са разним применама. Неки од тих слојева су деконволутивни, LSTM, attention, softmax... Коришћење разних слојева у саставу неуронских мрежа није једини начин да правимо сложеније и напредније моделе. Мреже можемо да структуирамо тако да уместо низа слојева имају графовску структуру.

3

Rust

3.1 О Rust-у

Rust је нов програмски језик са меморијским моделом који се разликује од оних на које можемо наичи у другим језицима и који му омогућује да буде веома ефикасан, а, такође, и веома сигуран. Неки језици користе GC(garbage collector), који стално прати која меморија се више не користи и ослобађа је(примери су C#, Java, Go), како би управљали меморијом. У другим језицима програмер мора експлицитно да алоцира и деалоцира меморију(примери су malloc, calloc, realloc и free у C, new и delete у C++). Руст има трећи приступ управљању меморијом. То је ownership. Ownership је скуп правила која су проверана на compile time-у. Та правила не само да неће имати runtime overhead који има GC него гарантују да ће програм бити меморијски сигуран за разлику од програма са мануелним управљањем меморије. Постоје 3 ownership правила и то су:

- Свака вредност има власника(owner-а)
- Ни једна вредност не може имати више од једног власника
- Када власник изађе из scope-а, вредност ће бити деалоцирана

3.2 Зашто Rust

Већина машинског учења се ради у Python-у. Постоји мноштво библиотека и оквира за машиншко учење и скоро сви имају API(Application Programming Interface) за рад у Python-у. Ово је вероватно зато сто је осим библиотека за машиншко учење Python-у доступан велики број других библиотека које су корисне за машиншко учење. Такође, Python је језик који је веома брз за развој софтвера и синтакса му је таква да свако може донекле да је разуме, без обзира

на знање Python-а. Међутим, Python је ради читљивости и брзине развоја жртвовао ефикасност. Иако се скоро све библиотеке за машиншко учење користе у Python-у, оне се најчешће пису у C++. C++ нема GC и за разлику од Pythona C++ има статички type system. Ово C++-у омогућава да буде веома брз. Иако је C++ type system статички исто као и Rust-ов, Rust-ове compile time провере су много обухватније и спречавају разне врсте недефинисаног понашања које се може појавити у C++-у. Чак и поред повећане сигурности Rust не губи на ефикасности. Rust, такође, има zero-cost абстракције које омогућавају корисницима да пишу код високог нивоа не губећи на ефикасности(један пример овога су итератори). Ове особине омогућују Rust-у да се такмичи не само са C++-ом већ и са Python-ом, јер док Rust са једне стране има ефикасност C++-а, он такође има абстракције вишег нивоа због којих донекле може да се пореди са Python-ом по читљивости. Rust ће увек бити ефикасан, а Python само онда кад се у позадини налази библиотека писана у неком другом језику(C++ например).

4

Примери имплементације

Тензор је структура која представља n-димензиони низ бројева. Тензори се користе за пренос података између слојева. Тензор у позадини држи једнодимензиони низ али постоје методе којима се од низа индекса добија индекс елемента у низу. Тензор у себи садржи и информацију о свом облику у виду структуре Shape. Како је тензор генерички тип који може да садрзи било коју врсту броја потребно је дефинисати trait за заједничко понасање свих бројева и још један trait за floating-point бројеве. Ово је урађено кроз Number и Float.

```
1  #[derive(Debug)]
2  pub struct Tensor<T: Number> {
3      shape: Shape,
4      data: Vec<T>,
5  }
6
7  impl<T: Number> Tensor<T> {
8      pub fn new<S: Into<Shape>>(shape: S) -> Self {
9          let shape = shape.into();
10         Self {
11             data: vec![T::zero(); shape.get_capacity()],
12             shape,
13         }
14     }
15
16     pub fn new_initialised<S: Into<Shape>, F: FnMut(usize)
       -> T>(shape: S, mut f: F) -> Self {
17         let mut t = Tensor::new(shape);
18         t.for_each_mut(|i, _| f(i));
19         t
```

```
20      }
21
22      pub fn for_each<F: FnMut(usize, T)>(&self, mut f: F) {
23          let mut i = 0;
24          while i < self.data.len() {
25              f(i, self.data[i]);
26              i += 1;
27          }
28      }
29
30      pub fn for_each_mut<F: FnMut(usize, T) -> T>(&mut self,
    mut f: F) {
31          let mut i = 0;
32          while i < self.data.len() {
33              self.data[i] = f(i, self.data[i]);
34              i += 1;
35          }
36      }
37
38      pub fn clear(&mut self) {
39          self.for_each_mut(|_, _| T::zero())
40      }
41
42      pub fn get_shape(&self) -> &Shape {
43          &self.shape
44      }
45  }
46
47  impl<T: Float> Tensor<T> {
48      pub fn is_nan(&self) -> bool {
49          for i in &self.data {
50              if i.is_nan() {
51                  return true;
52              }
53          }
54          false
55      }
56  }
57
58  impl<T: Number> Clone for Tensor<T> {
59      fn clone(&self) -> Self {
```

```
60          Self {
61              shape: self.shape.clone(),
62              data: self.data.clone(),
63          }
64      }
65  }
66
67  impl<T: Number> Index<usize> for Tensor<T> {
68      type Output = T;
69
70      fn index(&self, index: usize) -> &Self::Output {
71          &self.data[index]
72      }
73  }
74
75  impl<T: Number> IndexMut<usize> for Tensor<T> {
76      fn index_mut(&mut self, index: usize) -> &mut Self::
        Output {
77          &mut self.data[index]
78      }
79  }
80
81  pub {\latin trait} IntoIndex {
82      fn get_index(&self) -> &[usize];
83  }
84
85  impl<const N: usize> IntoIndex for [usize; N] {
86      fn get_index(&self) -> &[usize] {
87          &self[..]
88      }
89  }
90
91  impl<const N: usize> IntoIndex for &[usize; N] {
92      fn get_index(&self) -> &[usize] {
93          &(*self)[..]
94      }
95  }
96
97  impl IntoIndex for &[usize] {
98      fn get_index(&self) -> &[usize] {
99          self
```

```
100     }
101 }
102
103 impl<T: Number, U: IntoIndex> Index<U> for Tensor<T> {
104     type Output = T;
105
106     fn index(&self, index: U) -> &Self::Output {
107         &self.data[self.shape.index(index.get_index())]
108     }
109 }
110
111 impl<T: Number, U: IntoIndex> IndexMut<U> for Tensor<T> {
112     fn index_mut(&mut self, index: U) -> &mut Self::Output {
113         &mut self.data[self.shape.index(index.get_index())]
114     }
115 }
116
117 impl<T: Number> AddAssign<&Self> for Tensor<T> {
118     fn add_assign(&mut self, rhs: &Self) {
119         assert_eq!(self.shape, rhs.shape);
120         for i in 0..self.data.len() {
121             self.data[i] += rhs.data[i];
122         }
123     }
124 }
125
126 impl<T: Number> SubAssign<&Self> for Tensor<T> {
127     fn sub_assign(&mut self, rhs: &Self) {
128         assert_eq!(self.shape, rhs.shape);
129         for i in 0..self.data.len() {
130             self.data[i] -= rhs.data[i];
131         }
132     }
133 }
134
135 impl<T: Number> MulAssign<T> for Tensor<T> {
136     fn mul_assign(&mut self, rhs: T) {
137         for i in 0..self.data.len() {
138             self.data[i] *= rhs;
139         }
140     }
```

```
}

impl<T: Number> DivAssign<T> for Tensor<T> {
    fn div_assign(&mut self, rhs: T) {
        for i in 0..self.data.len() {
            self.data[i] /= rhs;
        }
    }
}
```

```
#[derive(Debug)]
pub struct Shape {
    shape: Vec<usize>,
    capacity: usize,
}

impl Shape {
    pub fn new<A: AsRef<[usize]>>(shape: A) -> Self {
        let shape = shape.as_ref();
        Shape {
            shape: Vec::from(shape),
            capacity: {
                if shape.is_empty() {
                    0
                } else {
                    let mut c = 1;
                    for i in shape {
                        c *= i;
                    }
                    c
                }
            },
        }
    }

    pub fn index(&self, index: &[usize]) -> usize {
        if index.len() == 1 {
            return index[0];
        }
        let mut s = 0;
        let mut p = 1;
```

```
32          for i in 0..index.len() {
33              assert!(index[i] < self[i]);
34
35              s += p * index[i];
36              p *= self[i];
37          }
38          s
39      }
40
41      pub fn get_dimension(&self) -> usize {
42          self.shape.len()
43      }
44
45      pub const fn get_capacity(&self) -> usize {
46          self.capacity
47      }
48  }
49
50  impl Clone for Shape {
51      fn clone(&self) -> Self {
52          Self {
53              shape: self.shape.clone(),
54              capacity: self.capacity,
55          }
56      }
57  }
58
59  impl PartialEq for Shape {
60      fn eq(&self, other: &Self) -> bool {
61          self.shape == other.shape
62      }
63  }
64
65  impl Index<usize> for Shape {
66      type Output = usize;
67
68      fn index(&self, index: usize) -> &Self::Output {
69          &self.shape[index]
70      }
71  }
72
```

```
73 impl<U: AsRef<[usize]>> From<U> for Shape {
74     fn from(shape: U) -> Self {
75         Shape::new(shape)
76     }
77 }
```

Layer је trait који дефинише заједничко понашање свих слојева. Сваки слој има три битне одговорности. Прва је ажурирање интерног стања. Друга је да од улазног низа тензора врати излазни низ тензора, ово ћемо звати пролаз унапред. Трећа је да од улазног низа тензора врати вредности извода свакод излазног параметра од сваког интерног параметра као и од сваког улазног параметра(параметар је у овом контексту свака вредност у сваком од улазних и интерних тензора). Ово ћемо звати пролаз уназад. Међутим, овај приступ дизајнирању слоја није најоптималнији, већ је заправо много боље да се у пролазу уназад врати извод неке крајне функције губитка од сваког, и улазног и интерног, параметра слоја. Проблем је наћи начин за добијање свих извода, али тако да слој буде без знања о ономе сто се дешава његовим излазним подацима. Постоји неколико начина да се ово уради. Један је да у дефиницији метода нема улазних тензора, већ да се сматра да су улазни тензори они који су били прослеђени методу за пролаз унапред. Међутим, то није добро решење, јер не постоји начин да и метод за пролаз унапред и метод за пролаз уназад буду оптимизовани. Пролаз уназад је неоптимизован ако мора да рачуна неке ствари које се ефикасније рацунају у пролазу унапред. Ако би се пролаз уназад оптимизовао свалили бисмо део одговорности на пролаз унапред. Посто пролаз унапред може да се користи потпуно независно од пролаза уназад додатна одговорност ће имати негативан утицај на ефикасност, а неће доприносити ничему. Постоји решење за овај проблем и то је додавање још једнод метода који ће служити за пролаз унапред али само онда кад ће се након његовог позива звати и пролаз уназад. У овом слуцају је могуче да све ради оптимално али долази до загађења кода као и до потребе да се неко парцијанло стање памти унутар слоја. Други начин да се уради пролаз уназад је да дефиниција метода има два параметра. Први је улазни низ тензора, а други параметар је функција која ће од излазних вредност датог слоја дати изводе улазних параметара слојева чији су улази излази датог слоја. На овај начин решавамо се проблема са ефикасносцу, јер је сва одговорност на једном методу, који има само једну сврху. Такође, пролаз уназад постаје једна целина. Изван ове три одговорности слој има јос неке споредне, које слузе за враћање информација о самом слоју, али оне су мање битне.

```
1 pub {\latin trait} Layer {
2     type Input: Number;
```

```
3       type Internal: Number;
4       type Output: Number;
5
6       fn get_input_shapes(&self) -> Vec<&Shape>;
7
8       fn get_output_shapes(&self) -> Vec<&Shape>;
9
10      fn feed_forward(&self, inputs: &[&Tensor<Self::Input>])
    -> Vec<Tensor<Self::Output>>;
11
12      fn back_propagate<'s>(
13          &'s self,
14          inputs: &[&Tensor<Self::Input>],
15          get_output_derivatives: Box<
16              dyn FnOnce(Vec<&Tensor<Self::Output>>) -> Vec<
    Option<Tensor<Self::Output>>> + 's,
17          >,
18      ) -> (
19          Option<Vec<Tensor<Self::Input>>>,
20          Option<Vec<Tensor<Self::Internal>>>,
21      );
22
23      fn update(&mut self, deltas: Vec<Tensor<Self::Internal
    >>) {
24          assert!(deltas.is_empty())
25      }
26  }
```

Model је структура која у себи садржи слојеве и организује интеракцију међу њима. Модел корисницима омогућава да се уместо мануелног управља групом слојева и организовања токова података међу њима то ради аутомацки. Постоје две врсте модела, то су секвенцијални и графовски. Графовски модели у себи садрже прецизно дефинисан усмерен нециклицан граф у чијим чворовима се налазе слојеви, а чије гране представљају ток података између њих. Секвенцијални модел је специјални слуцај графовског модела у ком је граф који га описује само прост пут.

```
1 pub struct Model<T: Float> {
2       layers: Vec<Node<T>>,
3       inputs: Vec<(usize, usize)>,
4       outputs: Vec<(usize, usize)>,
```

```
5 }
6
7 impl<T: Float> Model<T> {
8     fn new(layers: Vec<Node<T>>) -> Self {
9         let mut inputs = Vec::new();
10        let mut outputs = Vec::new();
11        for i in 0..layers.len() {
12            for j in 0..layers[i].inputs.len() {
13                match layers[i].inputs[j] {
14                    InputDefinition::Input(_) => inputs.push
    ((i, j)),
15                    InputDefinition::Internal(p, q) => {
16                        assert!(p < i);
17                        if let OutputDefinition::None(Option
    ::Some((i1, j1)))
18                        | OutputDefinition::Output(Option::
    Some((i1, j1)))
19                        | OutputDefinition::OutputWithLoss(_
    , Option::Some((i1, j1))) =
20                        layers[p].outputs[q]
21                        {
22                            assert_eq!(i, i1);
23                            assert_eq!(j, j1);
24                        } else {
25                            panic!()
26                        }
27                    }
28                }
29            }
30            for j in 0..layers[i].outputs.len() {
31                if let OutputDefinition::Output(_) |
    OutputDefinition::OutputWithLoss(_, _) =
32                layers[i].outputs[j]
33                {
34                    outputs.push((i, j));
35                }
36                if let OutputDefinition::None(Option::Some((
    p, q)))
37                | OutputDefinition::Output(Option::Some((p,
    q)))
38                | OutputDefinition::OutputWithLoss(_, Option
```

```
    ::Some((p, q))) =
39                  layers[i].outputs[j]
40                  {
41                      assert!(p > i);
42                      if let InputDefinition::Internal(i1, j1)
    = layers[p].inputs[q] {
43                          assert_eq!(i, i1);
44                          assert_eq!(j, j1);
45                      } else {
46                          panic!()
47                      }
48                  }
49              }
50          }
51          Self {
52              layers,
53              inputs,
54              outputs,
55          }
56      }
57
58      pub fn builder() -> GraphBuilder<T> {
59          GraphBuilder::new()
60      }
61
62      fn get_input_shapes(&self) -> Vec<&Shape> {
63          self.inputs
64              .iter()
65              .map(|(i, j)| self.layers[*i].layer.
    get_input_shapes()[*j])
66              .collect::<Vec<_>>()
67      }
68
69      fn get_output_shapes(&self) -> Vec<&Shape> {
70          self.outputs
71              .iter()
72              .map(|(i, j)| self.layers[*i].layer.
    get_output_shapes()[*j])
73              .collect::<Vec<_>>()
74      }
75
```

```
76      fn feed_forward(&self, inputs: &[&Tensor<T>]) -> Vec<
    Tensor<T>> {
77          assert_eq!(self.inputs.len(), inputs.len());
78          let mut inputs = inputs.into_iter();
79
80          let mut node_outputs = Vec::<Option<Vec<Tensor<T
    >>>>::with_capacity(self.layers.len());
81          for node in &self.layers {
82              let mut node_inputs = Vec::with_capacity(node.
    inputs.len());
83              for i in &node.inputs {
84                  node_inputs.push(match i {
85                      InputDefinition::Input(_) => *inputs.
    next().unwrap(),
86                      InputDefinition::Internal(p, q) => &
    node_outputs[*p].as_ref().unwrap()[*q],
87                  });
88              }
89              let output = node.layer.feed_forward(node_inputs
    .as_slice());
90              node_outputs.push(Option::Some(output));
91          }
92
93          let mut outputs = Vec::new();
94          for (p, q) in &self.outputs {
95              let output = node_outputs[*p]
96                  .take()
97                  .unwrap()
98                  .into_iter()
99                  .nth(*q)
100                 .unwrap();
101             outputs.push(output);
102         }
103         outputs
104     }
105
106     fn back_pass(
107         &self,
108         inputs: &[&Tensor<T>],
109         expected: &[&Tensor<T>],
110     ) -> Vec<Option<Vec<Tensor<T>>>> {
```

```
111         fn pass<'s, T: Float>(
112             index: usize,
113             nodes: &'s [Node<T>],
114             mut inputs: &[&Tensor<T>],
115             mut expected: &[&Tensor<T>],
116             node_outputs: &mut [Option<Vec<&Tensor<T>>>],
117             node_input_derivatives: &mut [Option<Vec<Tensor<
    T>>>],
118             node_internal_derivatives: &mut [Option<Vec<
    Tensor<T>>>],
119         ) {
120             if index == nodes.len() {
121                 return;
122             }
123             let node = &nodes[index];
124             let mut node_inputs = Vec::with_capacity(node.
    inputs.len());
125             for i in &node.inputs {
126                 node_inputs.push(match i {
127                     InputDefinition::Input(_) => {
128                         let (first, rest) = inputs.
    split_first().unwrap();
129                         inputs = rest;
130                         *first
131                     }
132                     InputDefinition::Internal(p, q) => &
    node_outputs[*p].as_ref().unwrap()[*q],
133                 });
134             }
135             let (input_derivatives, internal_derivatives) =
    node.layer.back_propagate(
136                 node_inputs.as_slice(),
137                 Box::new(|output| {
138                     let mut node_outputs = node_outputs.
    to_vec();
139                     let node_outputs = node_outputs.
    as_mut_slice();
140                     node_outputs[index] = Option::Some(
    output);
141                     pass(
142                         index + 1,
```

```
143                         nodes,
144                         inputs,
145                         expected,
146                         node_outputs,
147                         node_input_derivatives,
148                         node_internal_derivatives,
149                     );
150                     let mut node_output_derivatives = Vec::
    with_capacity(node.outputs.len());
151                     for i in 0..node.outputs.len() {
152                         node_output_derivatives.push(match &
    node.outputs[i] {
153                             OutputDefinition::None(Option::
    Some((p, q)))
154                             | OutputDefinition::Output(
    Option::Some((p, q))) => {
155                                 node_input_derivatives[*p]
156                                     .as_mut()
157                                     .map(|it| std::mem::
    replace(&mut it[*q], Tensor::new([])))
158                             }
159                             OutputDefinition::OutputWithLoss
    (loss, _) => Option::Some(
160                                 loss.derivative(&
    node_outputs[index].as_ref().unwrap()[i], {
161                                     let (first, rest) =
    expected.split_first().unwrap();
162                                     expected = rest;
163                                     *first
164                                 }),
165                             ),
166                             _ => Option::None,
167                         });
168                     }
169
170                     node_output_derivatives
171                 }),
172             );
173             node_input_derivatives[index] =
    input_derivatives;
174             node_internal_derivatives[index] =
```

```
    internal_derivatives;
175         }
176
177         assert_eq!(self.inputs.len(), inputs.len());
178
179         let mut node_outputs = vec![Option::None; self.
    layers.len()];
180         let mut node_input_derivatives = vec![Option::None;
    self.layers.len()];
181         let mut node_internal_derivatives = vec![Option::
    None; self.layers.len()];
182         pass(
183             0,
184             &self.layers,
185             inputs,
186             expected,
187             &mut node_outputs,
188             &mut node_input_derivatives,
189             &mut node_internal_derivatives,
190         );
191
192         node_internal_derivatives
193     }
194
195     fn update(&mut self, deltas: Vec<Option<Vec<Tensor<T
    >>>>) {
196         assert_eq!(self.layers.len(), deltas.len());
197
198         let mut deltas = deltas.into_iter();
199         for i in &mut self.layers {
200             if let Option::Some(d) = deltas.next().unwrap()
    {
201                 i.layer.update(d);
202             }
203         }
204     }
205 }
206
207 pub struct GraphBuilder<T: Float> {
208     layers: Vec<PartialNode<T>>,
209 }
```

```
210
211 impl<T: Float> GraphBuilder<T> {
212     fn new() -> Self {
213         Self {
214             layers: Vec::new(),
215         }
216     }
217
218     pub fn add_node<B, L>(mut self, node: NodeBuilder<T, B,
    L>) -> Self
219         where B: LayerBuilder<Input=T, Internal=T, Output=T,
    Layer=L>,
220               L: Layer<Input=T, Internal=T, Output=T> + '
    static, {
221         for i in 0..node.inputs.len() {
222             if let InputDefinition::Internal(p, q) = &node.
    inputs[i] {
223                 let n = self.layers.len();
224                 self.layers[*p].connect(*q, (n, i));
225             }
226         }
227         self.layers.push(
228             node.build(|p, q| self.layers[p].
    get_output_shape(q))
229         );
230         self
231     }
232
233     pub fn build(self) -> Model<T> {
234         Model::new(self.layers.into_iter().map(|l| l.build()
    ).collect::<Vec<_>>())
235     }
236 }
```

Како дефинисање графа није увек једноставно, поготово графа који испуњава све задате услове, постоји структура GraphBuilder. Ова структура омогућава корисницима да на једноставан начин направе графовски модел који ће бити валидан. Структуре и trait-ови који омогуцавају то су Node, PartialNode, NodeBuilder и LayerBuilder. Корисник ће креирати NodeBilder у који ће сместити LayerBuilder. Додавањем NodeBuilder-а у GraphBuilder, он ће се претворити у PartialNode користећи информације о претходним PartialNode-овима. Након

што је додат последњи NodeBuilder корисник може да позове build метод у GraphBuilder-у који ће ParcialNode-ове да претвори у Node-ове и да направи графовски модел.

```
pub struct GraphBuilder<T: Float> {
    layers: Vec<PartialNode<T>>,
}

impl<T: Float> GraphBuilder<T> {
    fn new() -> Self {
        Self {
            layers: Vec::new(),
        }
    }

    pub fn add_node<B, L>(mut self, node: NodeBuilder<T, B, L>) -> Self
        where B: LayerBuilder<Input=T, Internal=T, Output=T, Layer=L>,
              L: Layer<Input=T, Internal=T, Output=T> + 'static, {
        for i in 0..node.inputs.len() {
            if let InputDefinition::Internal(p, q) = &node.inputs[i] {
                let n = self.layers.len();
                self.layers[*p].connect(*q, (n, i));
            }
        }
        self.layers.push(
            node.build(|p, q| self.layers[p].get_output_shape(q))
        );
        self
    }

    pub fn build(self) -> Model<T> {
        Model::new(self.layers.into_iter().map(|l| l.build()).collect::<Vec<_>>())
    }
}
```

```rust
pub struct Node<T: Float> {
    pub(crate) layer: Box<dyn Layer<Input=T, Internal=T, Output=T>>,
    pub(crate) inputs: Vec<InputDefinition<()>>,
    pub(crate) outputs: Vec<OutputDefinition<Box<dyn Loss<T>>, Option<(usize, usize)>>>,
}

impl<T: Float> Node<T> {
    pub fn new(
        layer: Box<dyn Layer<Input=T, Internal=T, Output=T>>,
        inputs: Vec<InputDefinition<()>>,
        outputs: Vec<OutputDefinition<Box<dyn Loss<T>>, Option<(usize, usize)>>>,
    ) -> Self {
        Self {
            layer,
            inputs,
            outputs,
        }
    }
}
```

```rust
pub(crate) struct PartialNode<T: Float> {
    layer: Box<dyn Layer<Input=T, Internal=T, Output=T>>,
    inputs: Vec<InputDefinition<()>>,
    outputs: Vec<OutputDefinition<Box<dyn Loss<T>>, ()>>,
    new_outputs: BTreeMap<usize, (usize, usize)>,
}

impl<T: Float> PartialNode<T> {
    pub(self) fn new<L: Layer<Input=T, Internal=T, Output=T> + 'static>(layer: L, inputs: Vec<InputDefinition<()>>, outputs: Vec<OutputDefinition<Box<dyn Loss<T>>, ()>>) -> Self {
        Self {
            layer: Box::new(layer),
            inputs,
            outputs,
            new_outputs: BTreeMap::new(),
```

```
15          }
16      }
17
18      pub(crate) fn connect(&mut self, q: usize, child: (usize
    , usize)) {
19          let valid = self.new_outputs.insert(
20              q,
21              child,
22          ).is_none();
23          assert!(valid);
24      }
25
26      pub(crate) fn get_output_shape(&self, q: usize) -> &
    Shape {
27          self.layer.get_output_shapes()[q]
28      }
29
30      pub(crate) fn build(self) -> Node<T> {
31          let PartialNode {
32              layer,
33              inputs,
34              outputs,
35              mut new_outputs,
36          } = self;
37          let final_outputs = outputs
38              .into_iter()
39              .enumerate()
40              .map(|(i, o)| o.build(new_outputs.remove(&i)))
41              .collect::<Vec<_>>();
42
43          Node::new(layer, inputs, final_outputs)
44      }
45  }
```

```
1 pub struct NodeBuilder<
2      's,
3      T: Float,
4      B: LayerBuilder<Input=T, Internal=T, Output=T, Layer=L>
    + 'static,
5      L: Layer<Input=T, Internal=T, Output=T> + 'static,
6  > {
```

```
7       layer: B,
8       pub(crate) inputs: Vec<InputDefinition<&'s Shape>>,
9       pub(crate) outputs: Vec<OutputDefinition<Box<dyn Loss<T
    >>, ()>>,
10  }
11
12  impl<
13      's,
14      T: Float,
15      B: LayerBuilder<Input=T, Internal=T, Output=T, Layer=L>,
16      L: Layer<Input=T, Internal=T, Output=T> + 'static,
17  > NodeBuilder<'s, T, B, L>
18  {
19      pub fn new(
20          layer: B,
21          inputs: Vec<InputDefinition<&'s Shape>>,
22          outputs: Vec<OutputDefinition<Box<dyn Loss<T>>, ()
    >>,
23      ) -> Self {
24          Self {
25              layer,
26              inputs,
27              outputs,
28          }
29      }
30
31      pub(crate) fn build<F: Fn(usize, usize) -> &'s Shape>(
32          self,
33          get_parent_shape: F,
34      ) -> PartialNode<T> {
35          let NodeBuilder {
36              layer,
37              inputs,
38              outputs,
39          } = self;
40          let mut input_shapes = Vec::with_capacity(inputs.len
    ());
41          let mut input_defs = Vec::with_capacity(inputs.len()
    );
42          inputs.into_iter().for_each(|i| {
43              let (shape, input_def) = match i {
```

```
44                  InputDefinition::Input(shape) => (shape,
    InputDefinition::Input(())),
45                  InputDefinition::Internal(p, q) => (
    get_parent_shape(p, q), InputDefinition::Internal(p, q))
46              };
47              input_shapes.push(shape);
48              input_defs.push(input_def);
49          });
50
51          PartialNode::new(
52              layer.build(input_shapes.as_slice()),
53              input_defs,
54              outputs,
55          )
56      }
57  }
```

```
1  pub trait LayerBuilder {
2      type Input: Number;
3      type Internal: Number;
4      type Output: Number;
5      type Layer: Layer<Input = Self::Input, Internal = Self::
    Internal, Output = Self::Output>
6          + 'static;
7
8      fn build(self, input_shapes: &[&Shape]) -> Self::Layer;
9  }
```

Постоји још неколико битних структура и trait-ова који су неопходни за ово библиотеку. То су Loss, Activation, Optimizer, Dataset, Trainer.

Activation је trait који описује заједничко понашање активационих функција. Loss је trait који описује заједничко понашање функција губитка(грешке). Optimizer је trait који описује заједничко понашање алгоритама за оптимизацију. Dataset је trait који описује заједничко понашање скупова података за учење и предвиђање. Trainer је структура која прима оптимизациони алгоритам и скуп података и користи се за тренирање(учење) модела.

5

Примери истренираних модела

Неуронске мреже имају примене у разним областима и у могућности су да обављају разне и веома комплеске задатке. Међутим, што је задатак комплексији то је потребно више времена и више рачунарске моћи. Назалост, мени ни једно од та два није доступно, тако да ћу приказати само примене неуронских мрежа на простијим примерима где за тренирање модела није потребно много ресурса.

Mnist је скуп слика ручно писаних и обелезених цифара. Садржи 60000 слика за тренирање и 10000 слика за проверу тачности модела. Све слике су црно-беле и димензија 28x28. За овај скуп података се може користити најобичнија неуронска мрежа, састављена од 3 Dense слоја са редом 16, 16, 10 величинама излазних тенсора. На овај начин се може добити тачност од 95%. Али ово је далеко од најбољег резултата који се може добити. Користећи CNN можемо постици тачност од 97%. Уколико зелимо још боље резултате можемо да почетни скуп предпроцесирамо и да га на тај начин проширимо.

Iris flower(перунике) је скуп података у којем су дате информације за 150 цветова који припадају у 3 сорте(класе). Потребно је на основу 4 дужине одредјених делова цвета одредити којој класи припада. Као и за mnist и за овај скуп можемо користити неуронску мрежу састављену само од Dense слојева, и то у овом слуцају могу да имају још мање излазних параметара. Користећи тај метод можемо да добијемо тачност од близу 98%.

Boston Housing је скуп података у ком се уместо класификације, као што је рађено у претходним примерима, ради регресија. У овом скупу је дато 14 података о 506 кућа и треба одредити медијану цене кућа на основу осталих 13 података.

Назалост не постоје скупови података који су довољно прости, а да захтевају несеквенцијалне структуре.

6

Планови за будућност

Без обзира на модуларност ове библиотеке и сирок дијапазон модела који се могу креирати, она није без недостатка. Међу овим недостацима постоје они за чије решавање или није потребно много кода или потребан код није претерано сложен. Али, постоје и они којима решења зависе од Rust-а и можда никад неће бити потпуно решиви.

6.1 Лако решиви недостаци

Један од тих недостатака је то што тренутно није подржана употреба графичких картица. У данашње време се прилично велики део машинског учења одвија на графичким картицама као и на другом специјализованом харверу. Ово је корисно јер су те платформе дизајниране да омогуће масивно паралелно процесирање, које може значајно да скрати време тренирања као и време предвиђања. Иако постоји решење за овај недостатак, оно захтева да се неки делови кода паралелизују(користећи неку библиотеку, којом можемо да изврсавамо код на графицкој картици). Compile time критеријуми за паралелни код у Rust-у су још строжији него за секвенцијални код, сто би значило да би део постојећег кода морао да се промени.

Још један недостатак библиотеке је мањак одређених слојева који постоје у другим библиотекама за машиншко учење. Конкретно, то су RNN(recurrent neural network) и LSTM(long short-term memory) слојеви, као и неки други. Овај недостатак је лако решити, јер захтева само додавање неколико структура.

Све познате библиотеке за учење имају на неки начин имплементирану серијализацију модела. Односно, у свакој од њих постоји начин да се модел сачува у неком формату на диску или да се пошаље преко мреже. Серијализација у Rust-у може да буде веома једноставна, како постоје многе библиотеке које се

специјализују за тачно то.

6.2 Недостаци чија решења зависе од развоја Rust-a

Постоје недостаци које можда није могуче решити због самог дизајна Rust-a. У свим познатим библиотекама за учење постоји Embedding слој, чији су улазни подаци цели бројеви, а излазни реални бројеви. Иако се Embedding слој може написати и нормално функционисати у Rust-у, он се не може користити у моделу. Модел захтева да сваки садржани слој има исти тип улаза и излаза и тренутно не постоји начин да се ово ограничење заобиђе у Rust-у, али како је Rust нов језик који је под активним развојем, овако нешто мозда буде омогучено у бодућност. За сад, међутим, постоји начин да се Embedding слој користи у моделу, али тако да му улазни подаци буду реални бројеви, а да их посматра као целе. Слојеви са различитим улазним и излазним типовима нису једини проблем који настаје директно од Rust-a. Још један такав је немогућност прављења глобалног генеричког array pool-a. При свакој креацији тенсора алоцира се један низ за његове податке и сваки пут кад тенсор изађе из scope-a тај низ се деалоцира. Корисно би било када бисмо уместо да алоцирамо и деалоцирамо те низове, ми њих складистили негде одакле можемо да их узмемо при поновном креирању тенсора. На овај начин бисмо могли да побољсамо ефкасност, највисе зато сто се сваки тенсор који је у излазу слоја креира за скаки пролаз, а најчешће траје само то прослеђивања у следећи слој. Међутим, за овако нешто потребан је глобални објекат који ће да складисти низове, али тај објекат мора да буде генерички, јер ће се примарно користити у структури Тенсор која је генеричка. Ово би било могуче кад би постојали статички генерици. Као и са претходним проблемом ово мозда постане могуче у будуцности, али за разлику од прослог проблема тренуто не постоји никакв начин да овако несто постигнемо.

Литература

[1] The Rust Programming Language
`https://doc.rust-lang.org/book/`.

[2] The Rust Reference
`https://doc.rust-lang.org/reference/`.

[3] Michael Nielsen: Neural Networks and Deep Learning
`http://neuralnetworksanddeeplearning.com`.