Trax

deep learning 6

Raoul Grouls, 7 juni 2022

Machine learning in Python

Imagine some cars to drive 100x slower

To do machine learning, we would ideally want code to:

- Be as **fast** as possible, especially with matrix multiplications
- Handle the **gradient** for us, in order to learn weights with back propagation
- Have an **ecosystem** for machine learning

In terms of speed, Python seems to be a bad choice.

Possible solutions

- Use Python libraries that have their backend in C/C++ (e.g. PyTorch or TensorFlow)
- Learn a faster language, that also supports autograd
- Use Python libraries with a backend in JAX

julia

GORGONIA

GO reference | version 0.9.17 | Build and Tests on Linux/amd64 passing | codecov 58% | go report A+ | stability unstable

JAX

Jax is fast

```python
import numpy as np

def selu(x, alpha=1.67, lambda_=1.05):
    return lambda_ * np.where(x > 0, x, alpha * np.exp(x) - alpha)

x = np.arange(1000000)
%timeit selu(x)
```

✓ 5.7s

```
7.01 ms ± 100 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```python
import jax
import jax.numpy as jnp

def selu(x, alpha=1.67, lambda_=1.05):
    return lambda_ * jnp.where(x > 0, x, alpha * jnp.exp(x) - alpha)

x = jnp.arange(1000000)
%timeit selu(x).block_until_ready()
```

✓ 1.8s

```
2.24 ms ± 105 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```python
selu_jit = jax.jit(selu)
# Warm up
selu_jit(x).block_until_ready()
%timeit selu_jit(x).block_until_ready()
```

✓ 5.6s

```
689 µs ± 2.21 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Jax has an ecosystem

https://project-awesome.org/n2cholas/awesome-jax

Libraries

- Neural Network Libraries
  - Flax - Centered on flexibility and clarity.
  - Haiku - Focused on simplicity, created by the authors of Sonnet at De
  - Objax - Has an object oriented design similar to PyTorch.
  - Elegy - A High Level API for Deep Learning in JAX. Supports Flax, Haik
  - Trax - "Batteries included" deep learning library focused on providing
  - Jraph - Lightweight graph neural network library.
  - Neural Tangents - High-level API for specifying neural networks of bot
  - HuggingFace - Ecosystem of pretrained Transformers for a wide range
  - Equinox - Callable PyTrees and filtered JIT/grad transformations => ne
- NumPyro - Probabilistic programming based on the Pyro library.
- Chex - Utilities to write and test reliable JAX code.
- Optax - Gradient processing and optimization library.
- RLax - Library for implementing reinforcement learning agents.
- JAX, M.D. - Accelerated, differential molecular dynamics.
- Coax - Turn RL papers into code, the easy way.
- Distrax - Reimplementation of TensorFlow Probability, containing probabilit
- cvxpylayers - Construct differentiable convex optimization layers.
- TensorLy - Tensor learning made simple.
- NetKet - Machine Learning toolbox for Quantum Physics.

Trax

TensorFlow 3.0

- In TensorFlow, there is a lot of baggage that it just has to carry because it's backward compatible.
- Trax is a Google project, intended to improve TensorFlow, breaking backward compatibility.
- The focus is clear code and speed; you can literally read Trax source-code and understand what's going on.

Trax — Deep Learning with Clear Code and Speed

pypi package 1.4.1 | issues 87 open | build failing | contributions welcome | License Apache 2.0 | chat on gitter

Trax is an end-to-end library for deep learning that focuses on clear code and speed. It is actively used and maintained in the Google Brain team. This

https://github.com/google/trax

Downsides

- Trax is fairly recent. You will not find that much examples as for PyTorch / TensorFlow
- Even though there is an ecosystem, it is smaller than the PyTorch / TensorFlow ecosystems
- You will need to dive into the source code; a lot of documentation is to be found there.

Trax is clean

Implementations of Adam

```python
def update(self, step, grads, weights, slots, opt_params):
  m, v = slots
  learning_rate = opt_params['learning_rate']
  weight_decay_rate = opt_params['weight_decay_rate']
  b1 = opt_params['b1']
  b2 = opt_params['b2']
  eps = opt_params['eps']
  m = (1 - b1) * grads + b1 * m  # First  moment estimate.
  v = (1 - b2) * (grads ** 2) + b2 * v  # Second moment estimate.
  mhat = m / (1 - b1 ** (step + 1))  # Bias correction.
  vhat = v / (1 - b2 ** (step + 1))
  new_weights = ((1 - weight_decay_rate) * weights - (
      learning_rate * mhat / (jnp.sqrt(vhat) + eps))).astype(weights.dtype)
  return new_weights, (m, v)
```

Trax

```python
def _resource_apply_sparse(self, grad, var, indices, apply_state=None):
  var_device, var_dtype = var.device, var.dtype.base_dtype
  coefficients = ((apply_state or {}).get((var_device, var_dtype))
                  or self._fallback_apply_state(var_device, var_dtype))

  # m_t = beta1 * m + (1 - beta1) * g_t
  m = self.get_slot(var, 'm')
  m_scaled_g_values = grad * coefficients['one_minus_beta_1_t']
  m_t = tf.compat.v1.assign(m, m * coefficients['beta_1_t'],
                            use_locking=self._use_locking)
  with tf.control_dependencies([m_t]):
    m_t = self._resource_scatter_add(m, indices, m_scaled_g_values)

  # v_t = beta2 * v + (1 - beta2) * (g_t * g_t)
  v = self.get_slot(var, 'v')
  v_scaled_g_values = (grad * grad) * coefficients['one_minus_beta_2_t']
  v_t = tf.compat.v1.assign(v, v * coefficients['beta_2_t'],
                            use_locking=self._use_locking)
  with tf.control_dependencies([v_t]):
    v_t = self._resource_scatter_add(v, indices, v_scaled_g_values)

  if not self.amsgrad:
    v_sqrt = tf.sqrt(v_t)
    var_update = tf.compat.v1.assign_sub(
        var, coefficients['lr'] * m_t / (v_sqrt + coefficients['epsilon']),
        use_locking=self._use_locking)
    return tf.group(*[var_update, m_t, v_t])
  else:
    v_hat = self.get_slot(var, 'vhat')
    v_hat_t = tf.maximum(v_hat, v_t)
    with tf.control_dependencies([v_hat_t]):
      v_hat_t = tf.compat.v1.assign(
          v_hat, v_hat_t, use_locking=self._use_locking)
    v_hat_sqrt = tf.sqrt(v_hat_t)
    var_update = tf.compat.v1.assign_sub(
        var,
        coefficients['lr'] * m_t / (v_hat_sqrt + coefficients['epsilon']),
        use_locking=self._use_locking)
    return tf.group(*[var_update, m_t, v_t, v_hat_t])
```

Tensorflow

```python
def _single_tensor_adam(params: List[Tensor],
                        grads: List[Tensor],
                        exp_avgs: List[Tensor],
                        exp_avg_sqs: List[Tensor],
                        max_exp_avg_sqs: List[Tensor],
                        state_steps: List[Tensor],
                        *,
                        amsgrad: bool,
                        beta1: float,
                        beta2: float,
                        lr: float,
                        weight_decay: float,
                        eps: float,
                        maximize: bool):

    for i, param in enumerate(params):

        grad = grads[i] if not maximize else -grads[i]
        exp_avg = exp_avgs[i]
        exp_avg_sq = exp_avg_sqs[i]
        step_t = state_steps[i]
        # update step
        step_t += 1
        step = step_t.item()

        bias_correction1 = 1 - beta1 ** step
        bias_correction2 = 1 - beta2 ** step

        if weight_decay != 0:
            grad = grad.add(param, alpha=weight_decay)

        # Decay the first and second moment running average coefficient
        exp_avg.mul_(beta1).add_(grad, alpha=1 - beta1)
        exp_avg_sq.mul_(beta2).addcmul_(grad, grad.conj(), value=1 - beta2)
        if amsgrad:
            # Maintains the maximum of all 2nd moment running avg. till now
            torch.maximum(max_exp_avg_sqs[i], exp_avg_sq, out=max_exp_avg_sqs[i])
            # Use the max. for normalizing running avg. of gradient
            denom = (max_exp_avg_sqs[i].sqrt() / math.sqrt(bias_correction2)).add_(eps)
        else:
            denom = (exp_avg_sq.sqrt() / math.sqrt(bias_correction2)).add_(eps)



        step_size = lr / bias_correction1
        param.addcdiv_(exp_avg, denom, value=-step_size)


def _multi_tensor_adam(params: List[Tensor],
                       grads: List[Tensor],
                       exp_avgs: List[Tensor],
                       exp_avg_sqs: List[Tensor],
                       max_exp_avg_sqs: List[Tensor],
                       state_steps: List[Tensor],
                       *,
                       amsgrad: bool,
                       beta1: float,
                       beta2: float,
                       lr: float,
                       weight_decay: float,
                       eps: float,
                       maximize: bool):

    if len(params) == 0:
        return

    # update steps
    torch._foreach_add_(state_steps, 1)

    if maximize:
        grads = torch._foreach_neg(tuple(grads))  # type: ignore[assignment]

    bias_correction1 = [1 - beta1 ** step.item() for step in state_steps]
    bias_correction2 = [1 - beta2 ** step.item() for step in state_steps]
    if weight_decay != 0:
        torch._foreach_add_(grads, params, alpha=weight_decay)

    torch._foreach_mul_(exp_avgs, beta1)
    torch._foreach_add_(exp_avgs, grads, alpha=1 - beta1)

    torch._foreach_mul_(exp_avg_sqs, beta2)
    torch._foreach_addcmul_(exp_avg_sqs, grads, grads, 1 - beta2)

    if amsgrad:
        # Maintains the maximum of all 2nd moment running avg. till now
        max_exp_avg_sqs = torch._foreach_maximum(max_exp_avg_sqs, exp_avg_sqs)  # type:

        # Use the max. for normalizing running avg. of gradient
        max_exp_avg_sq_sqrt = torch._foreach_sqrt(max_exp_avg_sqs)
        bias_correction_sqrt = [math.sqrt(bc) for bc in bias_correction2]
        torch._foreach_div_(max_exp_avg_sq_sqrt, bias_correction_sqrt)
        denom = torch._foreach_add(max_exp_avg_sq_sqrt, eps)
    else:
        exp_avg_sq_sqrt = torch._foreach_sqrt(exp_avg_sqs)
        bias_correction_sqrt = [math.sqrt(bc) for bc in bias_correction2]
        torch._foreach_div_(exp_avg_sq_sqrt, bias_correction_sqrt)
        denom = torch._foreach_add(exp_avg_sq_sqrt, eps)

    step_size = [(lr / bc) * -1 for bc in bias_correction1]
    torch._foreach_addcdiv_(params, exp_avgs, denom, step_size)
```

PyTorch

Trax is readable

Linear / Dense layer

Documentation

Warning/ Error

Linear with bias

Linear without bias

```
def forward(self, x):
  """Executes this layer as part of a forward pass through the model.

  Args:
    x: Tensor of same shape and dtype as the input signature used to
        initialize this layer.

  Returns:
    Tensor of same shape and dtype as the input, except the final dimension
    is the layer's `n_units` value.
  """
  if self._use_bias:
    if not isinstance(self.weights, (tuple, list)):
      raise ValueError(f'Weights should be a (w, b) tuple or list; '
                       f'instead got: {self.weights}')
    w, b = self.weights
    return jnp.dot(x, w) + b  # Affine map.
  else:
    w = self.weights
    return jnp.dot(x, w)  # Linear map.
```

Trax is readable

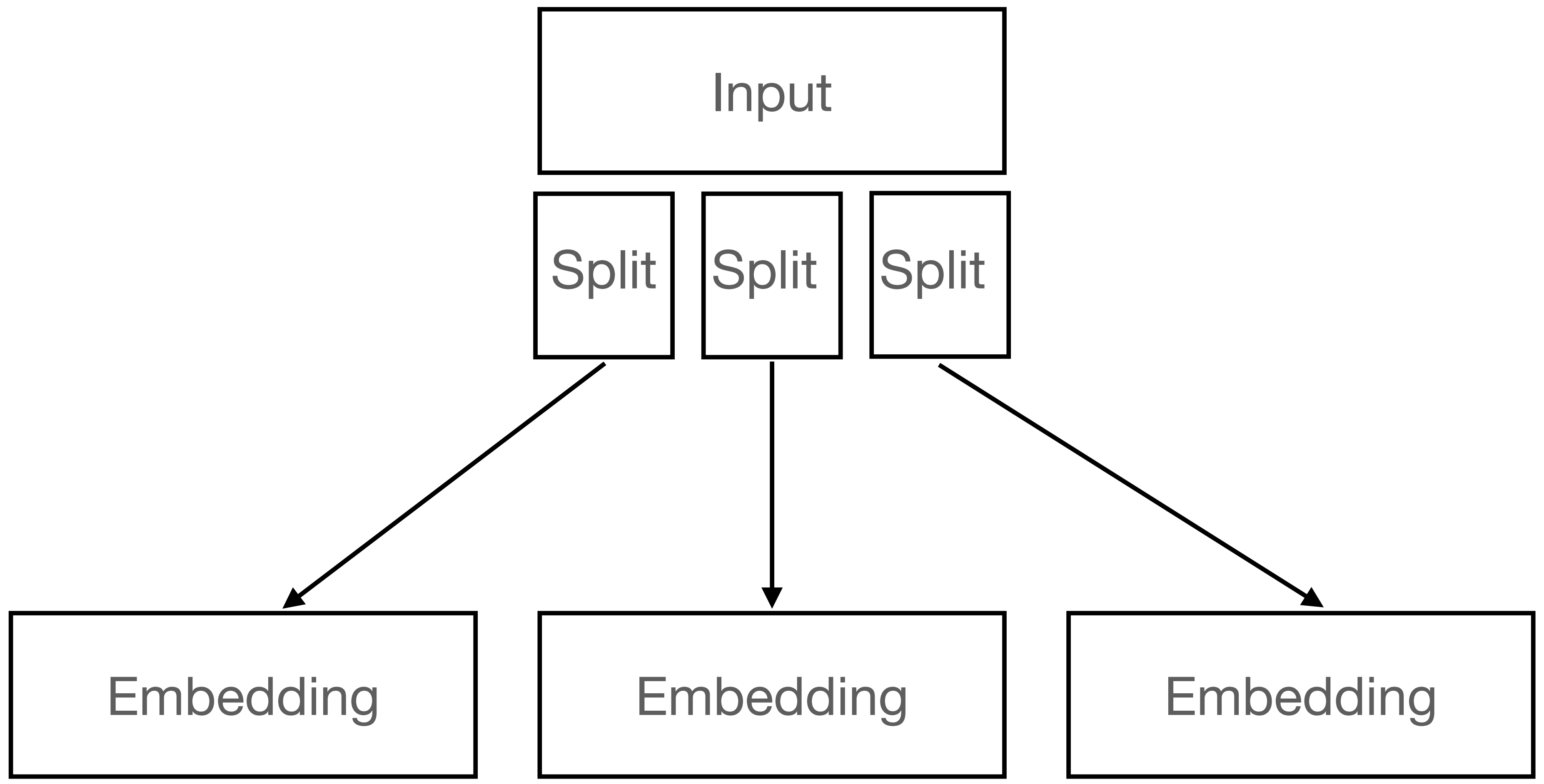
Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

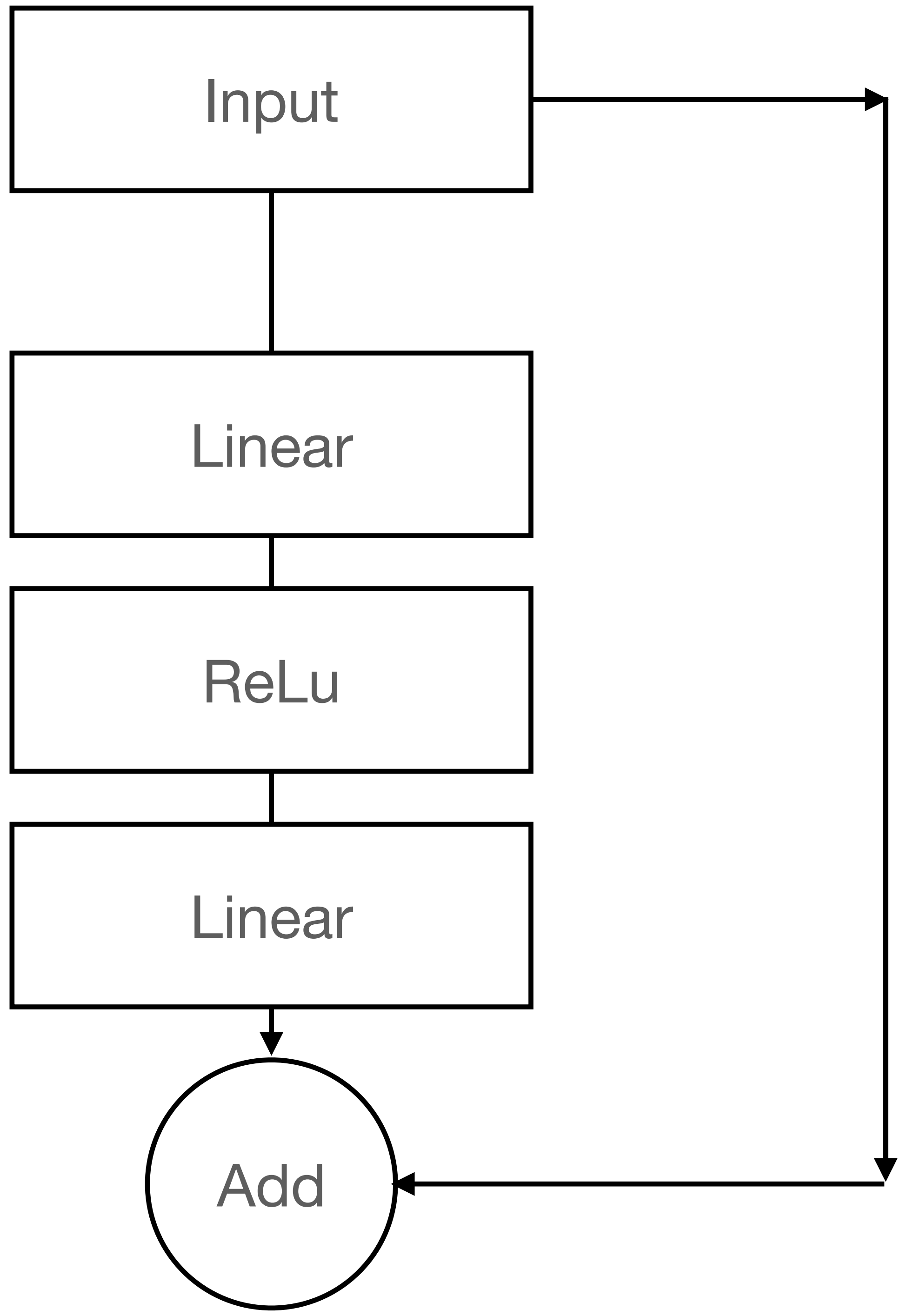
Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

```python
def update(self, step, grads, weights, slots, opt_params):
  m, v = slots
  learning_rate = opt_params['learning_rate']
  weight_decay_rate = opt_params['weight_decay_rate']
  b1 = opt_params['b1']
  b2 = opt_params['b2']
  eps = opt_params['eps']
  m = (1 - b1) * grads + b1 * m  # First  moment estimate.
  v = (1 - b2) * (grads ** 2) + b2 * v  # Second moment estimate.
  mhat = m / (1 - b1 ** (step + 1))  # Bias correction.
  vhat = v / (1 - b2 ** (step + 1))
  new_weights = ((1 - weight_decay_rate) * weights - (
      learning_rate * mhat / (jnp.sqrt(vhat) + eps))).astype(weights.dtype)
  return new_weights, (m, v)
```

Parallel flows

- In modern architectures, we will often want to implement models with parallel functions
- This can be done with plain Python, but it adds to the complexity
- Trax has multiple "combinators" designed to handle parallel flows





Parallel flows

Multi-embedding

```
emblist = [tl.Embedding(car, d_feature) for car in cardinality]
multiembedding = tl.Parallel(*emblist)
```

Trax implementation with a Parallel layer

Part of the 163 line long implementation in the PyTorch-Forecasting library

```
def init_embeddings(self):
    self.embeddings = nn.ModuleDict()
    for name in self.embedding_sizes.keys():
        embedding_size = self.embedding_sizes[name][1]
        if self.max_embedding_size is not None:
            embedding_size = min(embedding_size, self.max_embec
        # convert to list to become mutable
        self.embedding_sizes[name] = list(self.embedding_sizes[
        self.embedding_sizes[name][1] = embedding_size
```

```
input_vectors = {}
for name, emb in self.embeddings.items():
    if name in self.categorical_groups:
        input_vectors[name] = emb(
            x[
                ...,
                [self.x_categoricals.index(cat_name) for cat_name in self.categori
            ]
        )
    else:
        input_vectors[name] = emb(x[..., self.x_categoricals.index(name)])
```