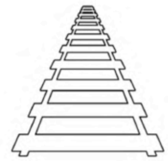
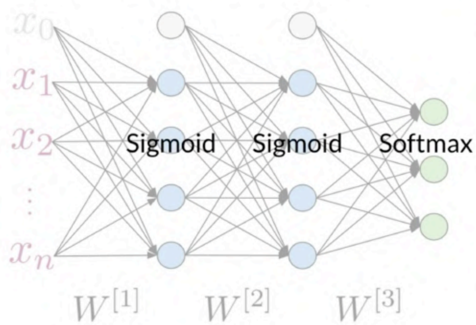
Outline

- Define a basic neural network using Trax
- Benefits of Trax



Neural Networks in Trax




```
from trax import layers as tl
Model = tl.Serial(
        tl.Dense(4),
        tl.Sigmoid(),
        tl.Dense(4),
        tl.Sigmoid(),
        tl.Dense(3),
        tl.Softmax())
```

Advantages of using frameworks

- Run fast on CPUs, GPUs and TPUs
- Parallel computing
- Record algebraic computations for gradient evaluation

Tensorflow Pytorch JAX

Summary

- Order of computation ⟶ Model in Trax
- Benefits from using frameworks

Trax makes programmers efficient

- Bottom-up clean re-design
- Easy to debug, you can read the code
- Full models with dataset bindings included
- Main models regression-tested daily

Is there a price to pay?

- Backwards compatibility

Trax runs code fast

- Designed to use a JIT compiler with JAX and XLA
- JAX: fastest Transformer in MLPerf 2020 (JAX: 0.26, TF: 0.35, pyTorch: 0.62)
- No code changes at all between CPU, GPU and TPU, preemptible training
- Tested with TPUs on colab too:

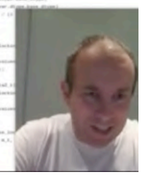
GPU (8x V100)	on-demand: $19.84 / hour	preemptible: $5.92 / hour
TPU (8x v3)	on-demand: $8 / hour	preemptible: $1.40 / hour

Keras

Pytorch

Adam Optimizer

$m_0 \leftarrow 0$ (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

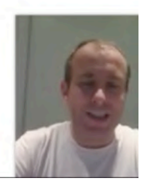


Slide 8 Q&A Notes Pointer Captions Tips EXIT

Trax Adam

```python
class Adam(Optimizer):
  """Adam optimizer."""

  def init(self, param):
    m = np.zeros_like(param)
    v = np.zeros_like(param)
    return m, v

  def update(self, step, grads, param, slots, opt_params):
    m, v = slots
    learning_rate, b1, b2, eps = opt_params
    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))
    param = param - (
        learning_rate * mhat / (np.sqrt(vhat) + eps)).astype(param.dtype)
    return param, (m, v)
```

$m_0 \leftarrow 0$ (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



Slide 9 Q&A Notes Pointer Captions Tips EXIT

Reading: (Optional) Trax and JAX, docs and code

Official Trax documentation maintained by the Google Brain team:

https://trax-ml.readthedocs.io/en/latest/

Trax source code on GitHub:

https://github.com/google/trax

JAX library:

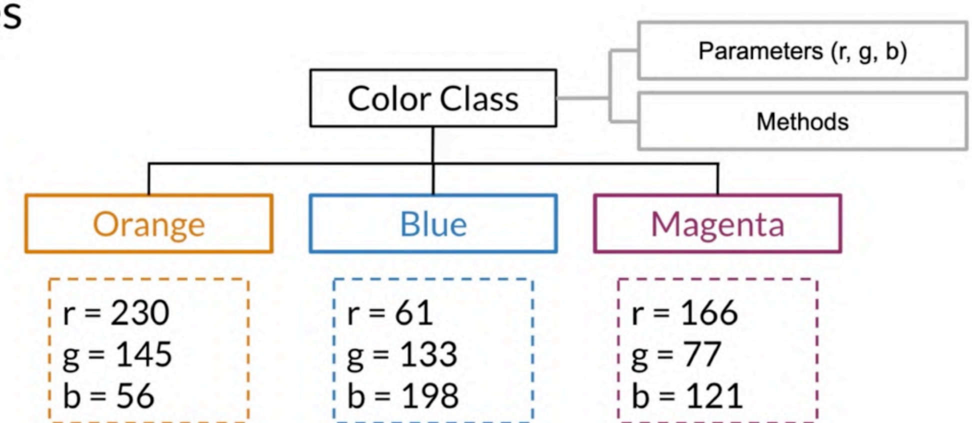
https://jax.readthedocs.io/en/latest/index.html

Layers

Outline

- How classes work and their implementation

Classes



Classes in Python

```python
class MyClass(Object):
    def __init__(self, y):
        self.y = y
    def my_method(self,x):
        return x + self.y

    def __call__(self, x):
        return self.my_method(x)
```

```python
f = MyClass(7)
```

Classes in Python

```python
class MyClass(Object):
    def __init__(self, y):
        self.y = y
    def my_method(self,x):
        return x + self.y
    def __call__(self, x):
        return self.my_method(x)
```

```python
f = MyClass(7)
print(f(3))
```

```
10
```

Subclasses

```python
class MyClass(Object):

    def __init__(self,y):
        self.y = y

    def my_method(self,x):
        return x + self.y

    def __call__(self,x):
        return self.my_method(x)
```

```python
class SubClass(MyClass):
```

Subclasses

```python
class MyClass(Object):

    def __init__(self,y):
        self.y = y

    def my_method(self,x):
        return x + self.y

    def __call__(self,x):
        return self.my_method(x)
```

Inherit parent MyClass

```python
class SubClass(MyClass):
    def my_method(self,x):
        return x + self.y**2
```

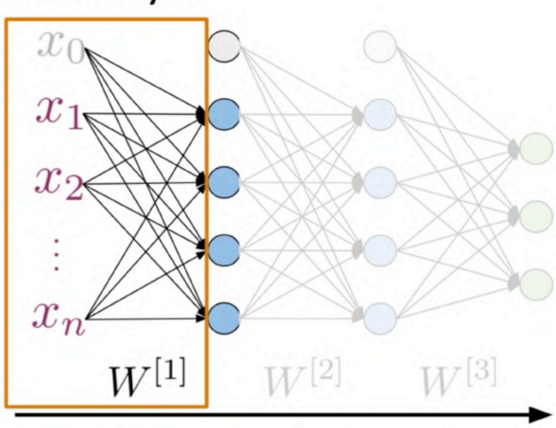
```python
f = SubClass(7)

print(f(3))
```

52

Outline

- Dense and ReLU layers

Dense Layer

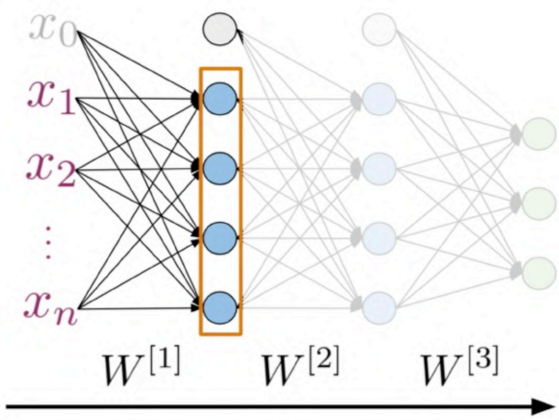


Fully connected layer

$$z^{[i]} = \boxed{W^{[i]}} a^{[i-1]}$$

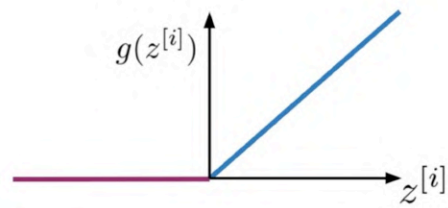
Trainable parameters

ReLU Layer



ReLU = Rectified linear unit

$$a^{[i]} = g(z^{[i]})$$
$$g(z^{[i]}) = \max(0, z^{[i]})$$

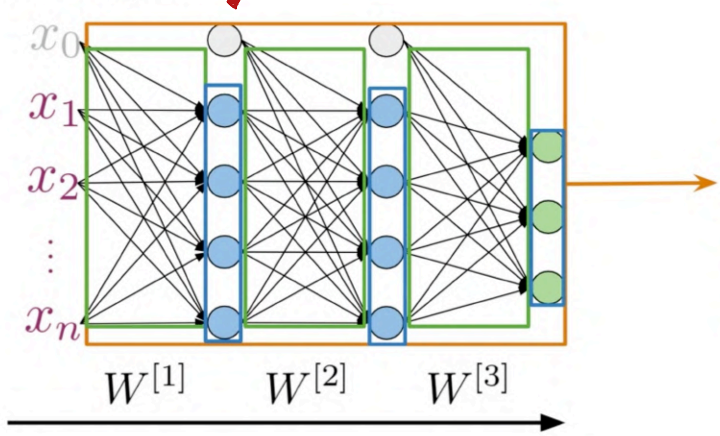


Summary

- Dense Layer $\longrightarrow$ $z^{[i]} = W^{[i]} a^{[i-1]}$
- ReLU Layer $\longrightarrow$ $g(z^{[i]}) = \max(0, z^{[i]})$

Think of this new serial layer as your whole neural network model in one layer.

Serial Layer



Composition of layers in *serial* arrangement

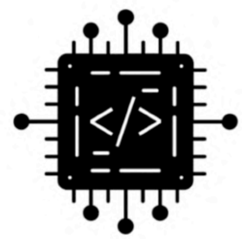
- Dense Layers
- Activation Layers

Summary

- Serial layer is a composition of sublayers

Outline

- Embedding layer
- Mean layer



Embedding Layer

Vocabulary	Index		
I	1	0.020	0.006
am	2	-0.003	0.010
happy	3	0.009	0.010
because	4	-0.011	-0.018
learning	5	-0.040	-0.047
NLP	6	-0.009	0.050
sad	7	-0.044	0.001
not	8	0.011	-0.022

Trainable weights

Vocabulary
x
Embedding

Mean Layer

Tweet: I am happy

Vocabulary	Index		
I	1	0.020	0.006
am	2	-0.003	0.010
happy	3	0.009	0.010

0.020	0.006
-0.003	0.010
0.009	0.010

Mean of the word embeddings

0.009
0.009

No trainable parameters

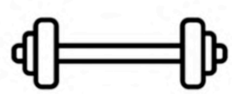
If you had padded vectors representing your tweets, you could unroll this matrix and feed its values to the next layer on the neural network. But in doing this, you might end up with lots of parameters to train. As an alternative, you could just take the mean of each feature from the embedding, and that's exactly what the mean layer does in tracks. After the mean layer, you will have the same number of features as you're embedding size.

Summary

- Embedding is trainable using an embedding layer
- Mean layer gives a vector representation

Outline

- Computing gradients in trax
- Training using grad()



Computing gradients in Trax

$$f(x) = 3x^2 + x$$

$$\boxed{\frac{\delta f(x)}{\delta x}} = 6x + 1$$

Gradient

```
def f(x):
    return 3*x**2 + x

grad_f = trax.math.grad(f)
```

Returns a function

Training with grad()

```
y = model(x)
grads = grad(y.forward)(y.weights,x)
```

In a loop

```
    weights -= alpha*grads
```

Gradient Descent

Forward and Back-propagation

Summary

- grad() allows much easier training
- Forward and backpropagation in one line!