Custom Layers and Loss Functions

Sept. 24, 2020

David Lawrence -- JLab

Motivation

Deep Learning has enabled a revolution:

Generate large, complex functions implementing many subtle correlations without having to develop a detailed mathematical model.

Black boxes are now *very* easy to build

Detailed understanding of all aspects of every tool is no longer required*

**you still have to understand what the black box does, even if you don't know how it does it.*

Motivation

Downside:

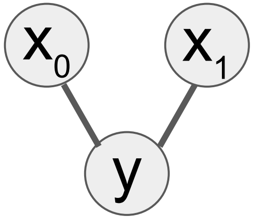
Simple, non-linear functions may take millions of parameters to mimic using standard Sequential networks and activation functions.

OK if your project requires infrequent training and inference.

Not OK if you deal with PB of data that must be run through a model quickly

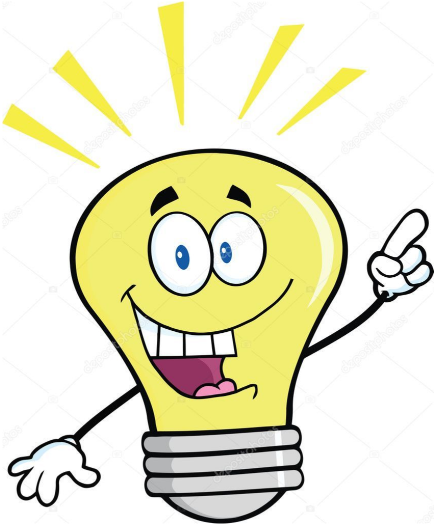
Motivation

What if you know something about the functional form that could be applied to your problem?



$$y = w_0 x_0 + w_1 x_1$$ easy!

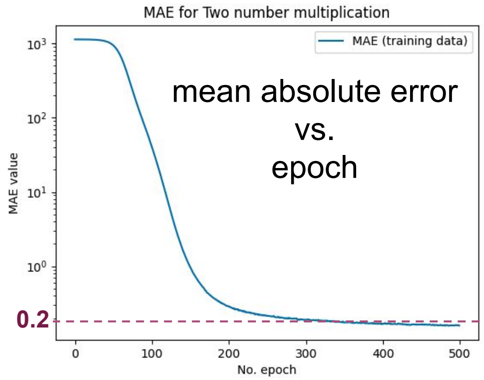
$$y = w_0 (x_0 * x_1)$$ ok, I think there's a way to do this



$$y = x_0^{w_0} * x_0^{w_1} + w_2 x_0 \log(x_1) - w_3 \csc(x_{01}/x_0) + \ldots$$ ummmm ....

Toy problem: Multiply 2 numbers $y = x_0^{w_0} x_1^{w_1}$

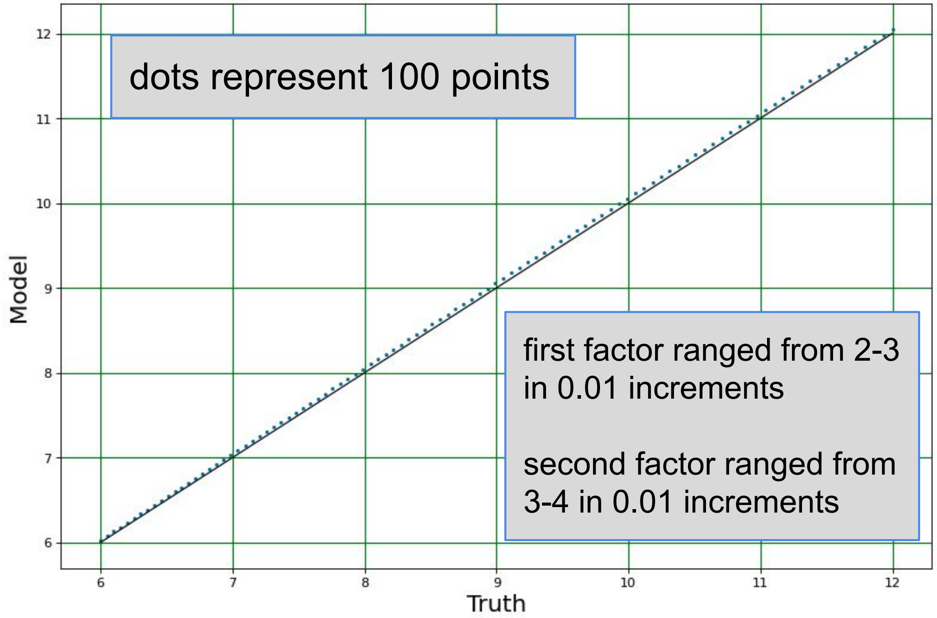
- Create a "deep" Sequential model with several Dense layers that alternate using "linear" and "tanh" functions.
- Train on dataset with inputs between -10 and +10 in 0.1 increments *(answers between -100 and +100)*
- Train 500 epochs with batch size 1000
- Use GPUs to train x1000 faster

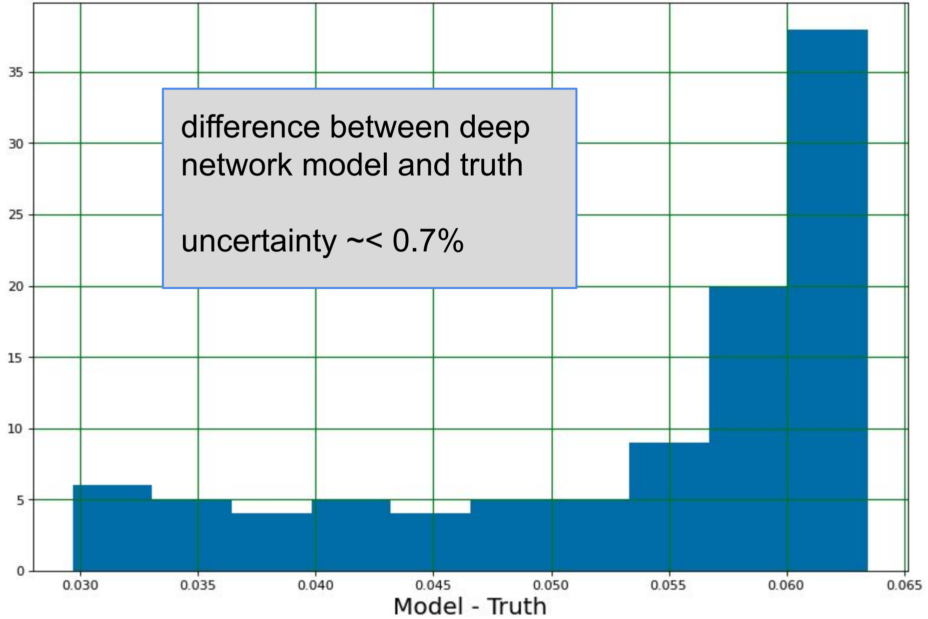


Layer (type)	Output Shape	Param #
waveform (InputLayer)	[(None, 2)]	0
top_layer1 (Flatten)	(None, 2)	0
common_layer1 (Dense)	(None, 1000)	3000
common_layer2 (Dense)	(None, 1000)	1001000
common_layer3 (Dense)	(None, 1000)	1001000
common_layer4 (Dense)	(None, 1000)	1001000
common_layer5 (Dense)	(None, 1000)	1001000
common_out1 (Dense)	(None, 1000)	1001000
common_out2 (Dense)	(None, 1000)	1001000
common_out3 (Dense)	(None, 200)	200200
outputs (Dense)	(None, 1)	201

Total params: 6,209,401
Trainable params: 6,209,401
Non-trainable params: 0

Deep Network vs. Truth of 2 number multiplication





Model could be modified and training tuned and extended ….

Custom layer with Keras

```python
#----------------------------------------------------
# ProductLayer
#----------------------------------------------------
# This defines a layer that takes the product of the inputs,
# each raised to the power of its weight. The trainable
# parameter can be set to False to make it non-trainable.
# n.b. If you make this trainable, the inputs cannot be
# negative numbers!
# See details on this in the following cell.
class ProductLayer(tf.keras.layers.Layer):
    def __init__(self, units=1, trainable=True, initial_exponent=2.01):
        super(ProductLayer, self).__init__()
        self.units            = units
        self.trainable        = trainable
        self.initial_exponent = initial_exponent

    def build(self, input_shape):
        print('input_shape='+str(input_shape))
        myinitializer    = tf.keras.initializers.Constant(self.initial_exponent)
        self.w = self.add_weight(
            shape       = (self.units, input_shape[-1]),
            initializer = myinitializer,
            trainable   = self.trainable,
        )
        self.b = self.add_weight(
            shape       = (self.units,),
            initializer = "zeros",
            trainable   = False
        )
```

`__init__()`:
Save any parameters your layer takes

`build()`:
Create/initialize weights (if any)

`call()`:
Define layer operations using weights

`get_config()`:
Return dictionary of layer configuration parameters for saving with model

```python
    def call(self, inputs):
        # inputs has shape (None, 2)
        # self.w has shape (1, 2)
        # tmp has shape (None, 2)
        # output has shape (None, 1)
        tmp = K.pow(inputs, self.w)
        myout = K.prod(tmp, keepdims=True, axis=1) + self.b
        print('inputs.shape: ' + str(inputs.shape))
        print('self.w.shape: ' + str(self.w.shape))
        print('   tmp.shape: ' + str(tmp.shape))
        print(' myout.shape: ' + str(myout.shape))
        return myout

    def get_config(self):
        config = super(ProductLayer, self).get_config()
        config.update({"units": self.units, "trainable": self.trainable, "initial_exponent": self.initial_exponent})
        return config
```

$$y = b_i + \prod_i x_i^{w_i}$$

Custom Layer with PyTorch

PyTorch

Labels defined as: $y = \sqrt{x_0 x_1^2}$

```python
# Extend nn.Module class and create a full custom layer that can be trained
class ProductLayer(nn.Module):
    # Initialization
    def __init__(self, input_size, output_size):
        super().__init__()
        self.input_size, self.output_size = input_size, output_size
        weights = torch.Tensor(output_size, input_size)
        self.weights = nn.Parameter(weights)
        # bias = torch.Tensor(output_size)
        # self.bias = nn.Parameter(bias)

        torch.nn.init.uniform_(self.weights, 0, 1)

    # Forward operations
    def forward(self, x):
        tmp = torch.pow(x, self.weights)
        return torch.prod(tmp)
```

```python
# Define a model that contains our custom layer
# Other layers can also be added but we don't really need them!
class BasicModel(nn.Module):
    def __init__(self):
        super().__init__()
        # self.conv = nn.Conv2d(16, 33, 3, stride=2)    # Example - how an in
        self.linear = ProductLayer(2, 1)                 # Input to our layer

    def forward(self, x):
        # x = self.conv(x)
        return self.linear(x)
```

```
===========================================================
Epoch:  0  MAE:  25.78905786835406
Weights:  tensor([[1.4386, 0.9737]])
===========================================================
Epoch:  1  MAE:  15.266186653409173
Weights:  tensor([[1.1287, 1.3200]])
===========================================================
Epoch:  2  MAE:  11.002496418743558
Weights:  tensor([[0.9659, 1.4993]])
===========================================================
Epoch:  3  MAE:  7.239018196825884
Weights:  tensor([[0.8063, 1.6748]])
===========================================================
Epoch:  4  MAE:  2.8084900026091315
Weights:  tensor([[0.5996, 1.9018]])
===========================================================
Epoch:  5  MAE:  0.11658574497252296
Weights:  tensor([[0.5004, 1.9999]])
---------- Average loss is too small, let me reduce learning rate now!! ----------------
===========================================================
Epoch:  6  MAE:  0.0024139598630172434
Weights:  tensor([[0.5000, 2.0000]])
---------- Average loss is too small, let me reduce learning rate now!! ----------------
---------- Average Loss is smaller than 0.005, Let's stop training further!! ------------
```

Model with Lambda layer

```
from tensorflow.keras.layers import Lambda
#-----------------------------------------------------
# MyProductLambda
#-----------------------------------------------------
def MyProductLambda(inputs):
    tmp = K.pow(inputs, (0.5, 2.0))
    return K.prod(tmp, keepdims=True, axis=1)

#-----------------------------------------------------
# DefineModelLambda
#-----------------------------------------------------
def DefineModelLambda():

    # Build the network model with 2 inputs and one output.
    inputs = Input(shape=(NINPUTS,), name='inputs')
    output = Lambda(MyProductLambda, output_shape=(1,))(inputs)
    model  = Model(inputs=inputs, outputs=output)

    opt = Adadelta(clipnorm=1.0)
    model.compile(loss='mse', optimizer=opt, metrics=['mae', 'mse', 'accuracy'])

    return model


model_lambda = DefineModelLambda()
```

Lambda Layers:

- No trainable weights

From Keras Documentation

The main reason to subclass `tf.keras.layers.Layer` instead of using a `Lambda` layer is saving and inspecting a Model. `Lambda` layers are saved by serializing the Python bytecode, whereas subclassed Layers can be saved via overriding their `get_config` method. Overriding `get_config` improves the portability of Models. Models that rely on subclassed Layers are also often easier to visualize and reason about.

Backend functions

- Custom loss and custom layer functions are *NOT* called for every set of inputs
- They are called *ONCE* to define a set of operations (**ops**)
- Keras/Tensorflow can then:
  - Take derivatives of operations
  - Optimize for the hardware the model runs on
- The backend functions allow one to build up a set of operations similar to how one builds a network from multiple layers

backend functions allow the system to do a whole lot of optimization beneath the surface

A simple backend function

tf.keras.backend.exp

View source on GitHub

Element-wise exponential.

View aliases

```
tf.keras.backend.exp(
    x
)
```

Arguments	
x	Tensor or variable.

Returns
A tensor.

dimension 0

dimension 1

$$\begin{pmatrix} A & B & C \\ D & E & F \end{pmatrix}$$

3x2

$$\Downarrow$$

$$\begin{pmatrix} e^A & e^B & e^C \\ e^D & e^E & e^F \end{pmatrix}$$

3x2

A slightly less simple backend function

tf.keras.backend.dot

TensorFlow 1 version | View source on GitHub

Multiplies 2 tensors (and/or variables) and returns a tensor.

View aliases

```
tf.keras.backend.dot(
    x, y
)
```

Arguments	
x	Tensor or variable.
y	Tensor or variable.

Returns
A tensor, dot product of **x** and **y**.

dimension 0

dimension 1

$$\underset{y\ (2x3)}{\begin{pmatrix} G & H \\ I & J \\ K & L \end{pmatrix}} \bullet \underset{x\ (3x2)}{\begin{pmatrix} A & B & C \\ D & E & F \end{pmatrix}}$$

$$\begin{pmatrix} (GA+HD) & (GB+HE) & (GC+HF) \\ (IA+JD) & (IB+JE) & (IC+JF) \\ (KA+LD) & (KB+LE) & (KC+LF) \end{pmatrix}$$

3x3

loops over ***last*** dimension in x and ***next-to-last*** dimension in y

A slightly less simple backend function

tf.keras.backend.dot

TensorFlow 1 version | View source on GitHub

Multiplies 2 tensors (and/or variables) and returns a tensor.

View aliases

```
tf.keras.backend.dot(
    x, y
)
```

Arguments	
x	Tensor or variable.
y	Tensor or variable.

Returns
A tensor, dot product of x and y.

```
# Testing dot vs batch_dot
import tensorflow.keras.backend as K

x = K.placeholder(shape=(3, 2))
y = K.placeholder(shape=(2, 3))
xy = K.dot(x,y)
print(xy.shape)
```

```
(3, 3)
```

```
x = K.placeholder(shape=(32, 28, 3))
y = K.placeholder(shape=(3, 4))
xy = K.dot(x,y)
print(xy.shape)
```

```
(32, 28, 4)
```

```
x = K.placeholder(shape=(2, 7, 3))
y = K.placeholder(shape=(6, 4, 3, 5))
xy = K.dot(x,y)
print(xy.shape)
```

```
(2, 7, 6, 4, 5)
```

loops over ***last*** dimension in `x` and ***next-to-last*** dimension in `y`

A slight problem: Data comes in batches

- During training, inputs and labels are presented in *batches*. The size of the batch may actually vary during training.
- The batch size is not known when defining the model and so is represented as ***None*** in the first dimension of the shape.
- This presents a problem when using ***dot***

```
x = K.placeholder(shape=(None, 8, 7, 6))
y = K.placeholder(shape=(None, 7, 5, 6, 2))
xy = K.dot(x,y)
print(xy.shape)
```

```
(None, 8, 7, None, 7, 5, 2)
```

output should only include batch size(i.e. ***None***) in the first dimension!

batch_dot to the rescue!

tf.keras.backend.batch_dot

TensorFlow 1 version | View source on GitHub

Batchwise dot product.

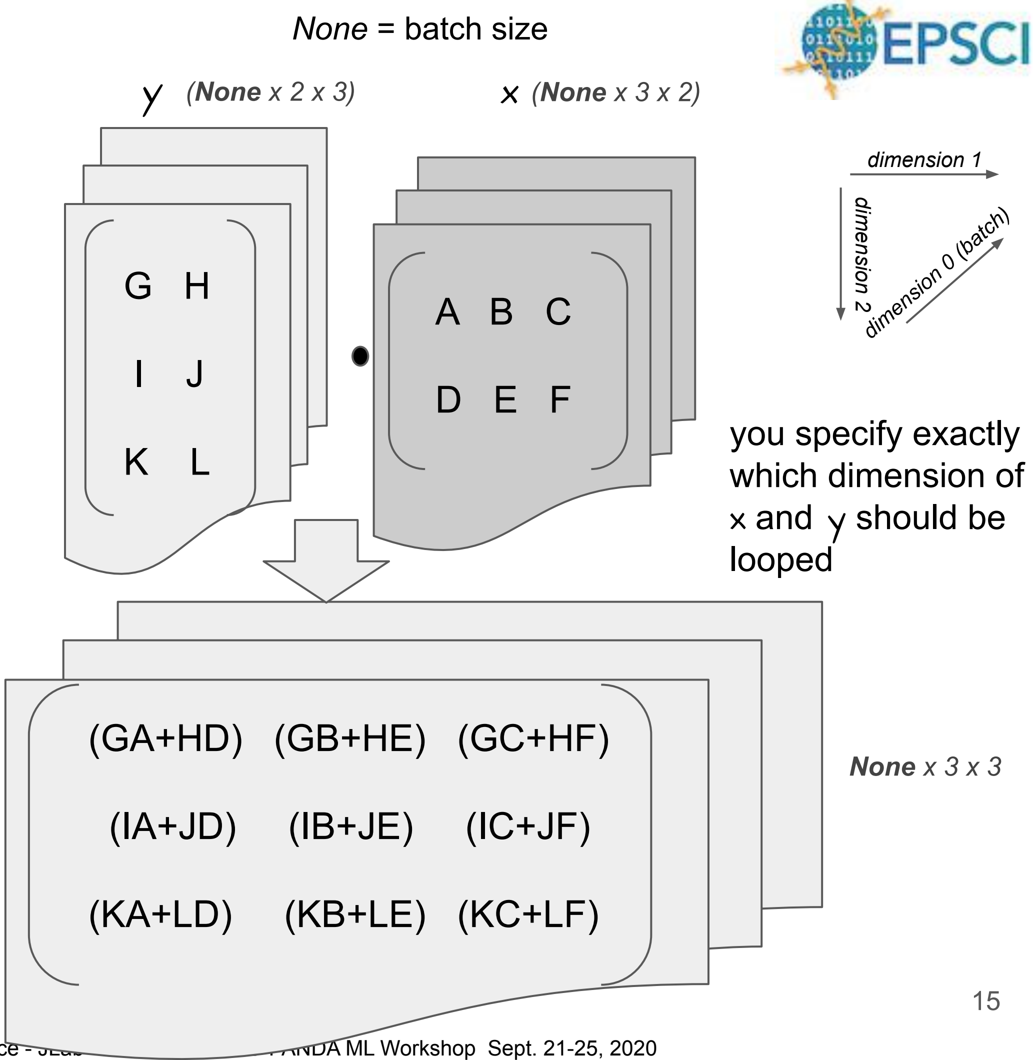
View aliases

```
tf.keras.backend.batch_dot(
    x, y, axes=None
)
```

`batch_dot` is used to compute dot product of `x` and `y` when `x` and `y` are data in batch, i.e. in a shape of `(batch_size, :)`. `batch_dot` results in a tensor or variable with less dimensions than the input. If the number of dimensions is reduced to 1, we use `expand_dims` to make sure that ndim is at least 2.

Arguments	
`x`	Keras tensor or variable with `ndim >= 2`.
`y`	Keras tensor or variable with `ndim >= 2`.
`axes`	Tuple or list of integers with target dimensions, or single integer. The sizes of `x.shape[axes[0]]` and `y.shape[axes[1]]` should be equal.

Returns
A tensor with shape equal to the concatenation of x's shape (less the dimension that was summed over) and y's shape (less the batch dimension and the dimension that was summed over). If the final rank is 1, we reshape it to `(batch_size, 1)`.



you specify exactly which dimension of x and y should be looped

batch_dot

tf.keras.backend.batch_dot

TensorFlow 1 version

View source on GitHub

Batchwise dot product.

View aliases

```
tf.keras.backend.batch_dot(
    x, y, axes=None
)
```

`batch_dot` is used to compute dot product of `x` and `y` when `x` and `y` are data in batch, i.e. in a shape of `(batch_size, :)`. `batch_dot` results in a tensor or variable with less dimensions than the input. If the number of dimensions is reduced to 1, we use `expand_dims` to make sure that ndim is at least 2.

Arguments	
x	Keras tensor or variable with `ndim >= 2`.
y	Keras tensor or variable with `ndim >= 2`.
axes	Tuple or list of integers with target dimensions, or single integer. The sizes of `x.shape[axes[0]]` and `y.shape[axes[1]]` should be equal.

Returns
A tensor with shape equal to the concatenation of x's shape (less the dimension that was summed over) and y's shape (less the batch dimension and the dimension that was summed over). If the final rank is 1, we reshape it to `(batch_size, 1)`.

dot

```
x = K.placeholder(shape=(None, 8, 7, 6))
y = K.placeholder(shape=(None, 7, 5, 6, 2))
xy = K.dot(x,y)
print(xy.shape)
```

```
(None, 8, 7, None, 7, 5, 2)
```

batch_dot

```
x = K.placeholder(shape=(None, 8, 7, 6))
y = K.placeholder(shape=(None, 7, 5, 6, 2))
xy = K.batch_dot(x,y)
print(xy.shape)
```

```
(None, 8, 7, 7, 5, 2)
```

```
x = K.placeholder(shape=(None, 8, 7, 6))
y = K.placeholder(shape=(None, 7, 5, 6, 2))
xy = K.batch_dot(x,y, axes=(2,1))
print(xy.shape)
```

```
(None, 8, 6, 5, 6, 2)
```

you can also specify which axes to loop over!

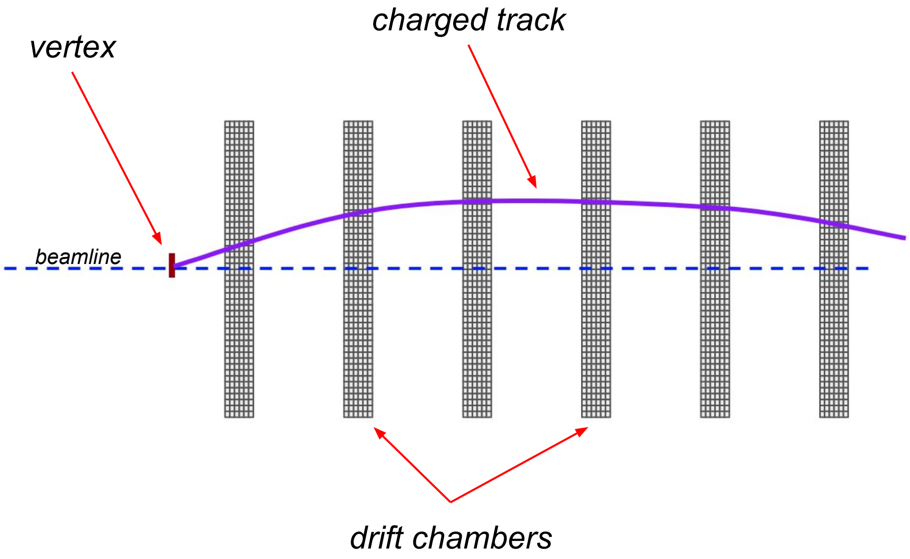
Example: Charged Particle Tracking

- Goal is to get 5 parameter state vector at the vertex
  (3-momentum + 2-position)

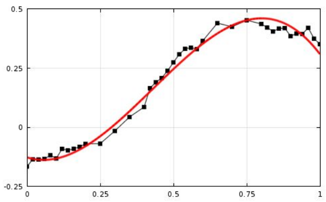
- Also need the covariance matrix
  (15 parameters)

- Most common solution is to use Kalman filter
  - Provides both state vector and covariance matrix

- Suppose we have a working Kalman filter solution, but want to develop an ML model that reproduces it.
  (surrogate model)



Model training = Curve fitting



mse is the same as

χ^2 minimization

i.e. Minimize number of σ's the model is from data by adjusting model parameters

σ_i represents uncertainty in the y_i values

loss labels model output model input

$$\chi^2 = \sum \frac{[y_i - f(x_i)]^2}{\sigma_i^2}$$

$$\chi_i^2 = \vec{\delta s_i}^{\mathsf{T}} \cdot C^{-1} \cdot \vec{\delta s_i}$$ *generalize*

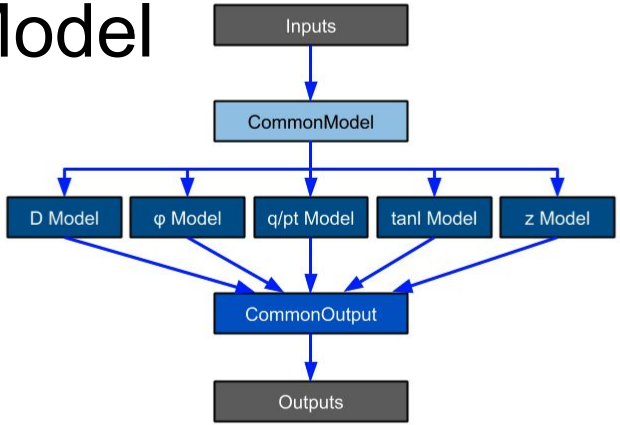
$$\vec{\delta s_i} = (\vec{s_i^{label}} - \vec{s_i^{model}})$$

$$\vec{s} = \begin{bmatrix} q/p_t \\ \phi \\ tanl \\ D \\ z \end{bmatrix}$$

$$\mathrm{C} = \begin{bmatrix} \sigma^2_{q/p_t} & \sigma_{q/p_t}\sigma_\phi & \sigma_{q/p_t}\sigma_{tanl} & \sigma_{q/p_t}\sigma_D & \sigma_{q/p_t}\sigma_z \\ \ddots & \sigma^2_\phi & \sigma_\phi\sigma_{tanl} & \sigma_\phi\sigma_D & \sigma_\phi\sigma_z \\ \ddots & \ddots & \sigma^2_{tanl} & \sigma_{tanl}\sigma_D & \sigma_{tanl}\sigma_z \\ \ddots & \ddots & \ddots & \sigma^2_D & \sigma_D\sigma_z \\ \ddots & \ddots & \ddots & \ddots & \sigma^2_z \end{bmatrix}$$

covariance matrix

Using custom loss to fit a Tracking Model




```python
# Here we build the network model.
# This model is made of multiple parts. The first handles the
# inputs and identifies common features. The rest are branches with
# each determining an output parameter from those features.
inputs         = Input(shape=(NINPUTS,), name='image_inputs')
commonmodel    = DefineCommonModel(inputs)
Dmodel         = DefinePhiModel(          commonmodel )
phimodel       = DefineDModel(            commonmodel )
q_over_ptmodel = Define_q_over_pt_Model( commonmodel )
tanlmodel      = Define_tanl_Model(       commonmodel )
zmodel         = DefineZModel(            commonmodel )
commonoutput   = DefineCommonOutput([Dmodel, phimodel, q_over_ptmodel, tanlmodel, zmodel])

# Custom loss function requires additional inputs (inverse covariance matrix)
# We must use the "add_loss" method to get around restrictions that prevent
# us from just passing them in as extra labels. See:
# https://stackoverflow.com/questions/62154660/keras-valueerror-dimensions-must-be-equal-ho
input_true     = Input(shape=(5,), name='state_true')
input_incov    = Input(shape=(5*5,), name='invcov')
all_inputs     = [inputs, input_incov, input_true]

model          = Model(inputs=all_inputs, outputs=commonoutput)
```

```python
final_model.add_loss(customLoss( input_true, commonoutput, input_incov ) )
final_model.compile(loss=None, optimizer=opt, metrics=['mae', 'mse', 'accuracy'])
```

Total "inputs" includes hit info., inverse covariance matrix, and labels

Custom Loss function needs these same "inputs"

$$\chi_i^2 = \vec{\delta s_i}^{\mathsf{T}} \cdot C^{-1} \cdot \vec{\delta s_i}$$

```python
#-----------------------------------------
# Define custom loss function
def customLoss(y_true, y_pred, invcov):

    batch_size = tf.shape(y_pred)[0]  # n.b. y_pred.shape[0] will not work for some reason in tf1
    print('y_pred shape: ' + str(y_pred.shape) )  # y_pred shape is (batch, 5)
    print('y_true shape: ' + str(y_true.shape) )  # y_true shape is (batch, 5)
    print('invcov shape: ' + str(invcov.shape) )  # inconv shape is (batch, 25)

    y_pred = K.reshape(y_pred, (batch_size, 5,1)) # y_pred  shape is now (batch, 5,1)
    y_true = K.reshape(y_true, (batch_size, 5,1)) # y_state shape is now (batch, 5,1)
    invcov = K.reshape(invcov, (batch_size, 5,5)) # invcov  shape is now (batch, 5,5)

    # n.b. we must use tf.transpose here an not K.transpose since the latter does not allow perm argument
    invcov = tf.transpose(invcov, perm=[0,2,1])     # invcov shape is now (batch, 5,5)

    # Difference between prediction and true state vectors
    y_diff = y_pred - y_true

    # n.b. use "batch_dot" and not "dot"!
    y_dot = K.batch_dot(invcov, y_diff)              # y_dot shape is (batch,5,1)
    y_dot = K.reshape(y_dot, (batch_size, 1, 5))     # y_dot shape is now (batch,1,5)
    y_loss = K.batch_dot(y_dot, y_diff)              # y_loss shape is (batch,1,1)
    y_loss = K.reshape(y_loss, (batch_size,))        # y_loss shape is now (batch)
    return y_loss
```

**most of the effort is in getting the shapes right!*

Testing the custom Loss function

```
#---------------------------------------------
# Test loss function
y_pred_vals = [1.0, 2.0, 3.0, 4.0, 5.0]
y_true_vals = [1.1, 2.1, 3.1, 4.1, 5.1]
inconv_vals = np.arange(0.1, 2.6, 0.1).tolist()

# Make of batch of 3, but just use the same values for each
y_pred = K.variable([y_pred_vals, y_pred_vals, y_pred_vals])
y_true = K.variable([y_true_vals, y_true_vals, y_true_vals])
inconv = K.variable([inconv_vals, inconv_vals, inconv_vals])

loss = K.eval(customLoss(y_true, y_pred, inconv))
print('loss shape: '    + str(loss.shape)    )
print(loss)
```

25 floating point values

Create and EVALUATE a custom loss Tensor

```
y_pred shape: (3, 5)
y_true shape: (3, 5)
invcov shape: (3, 25)
loss shape: (3,)
[0.32499945 0.32499945 0.32499945]
```

- Test loss function using a few known values
- Create backend variables to hold the values just like the layers do in the full model
- Pass in a small batch to ensure loss function handles batch dimension correctly

Jefferson Lab

EPSCI

Summary

Custom Layers allow one to insert specific knowledge of mathematical forms into the model, potentially relaxing how deep the architecture needs to be

Custom Loss functions allow specific knowledge of the uncertainties to be applied while training regression models

Links to notebooks:

https://github.com/faustus123/Jupyter/blob/master/2020.08.15.CustomLayers/2020.08.15.Multiplication.ipynb

https://github.com/faustus123/Jupyter/blob/master/2020.08.15.CustomLayers/2020.08.25.Mulitplication_customLayer.ipynb

Backup Slides

Alternative Way to Pass Extra Arguments to CustomLoss

```
#----------------------------------------------
# Define custom loss function
def customLoss2(y_true, y_pred):

    batch_size = tf.shape(y_pred)[0]  # n.b. y_pred.shape[0] will not work for some reason in tf1
    print('y_pred shape: ' + str(y_pred.shape) )  # y_pred shape is (batch, 5)
    print('y_true shape: ' + str(y_true.shape) )  # y_true shape is (batch, 49)
    print('y_pred type: ' + str(type(y_pred) ) )  # y_pred shape is (batch, 5)
    print('y_true type: ' + str(type(y_true) ) )  # y_true shape is (batch, 49)

    # Note that y_pred only has the 5 state vector parameters while y_true contains
    # all of the labels (event, state vector, covariance matrix, inverse cov., ...)
    # We peel off the state vector and inverse covariance here which are the parts
    # we need.
    y_state = y_true[:,1:6]                        # y_state shape is now (batch, 5)
    invcov  = y_true[:,21:46]                      # invcov  shape is now (batch, 25)

    y_pred  = K.reshape(y_pred,  (batch_size, 5,1)) # y_pred  shape is now (batch, 5,1)
    y_state = K.reshape(y_state, (batch_size, 5,1)) # y_state shape is now (batch, 5,1)
    invcov  = K.reshape(invcov,  (batch_size, 5,5)) # invcov  shape is now (batch, 5,5)

    # n.b. we must use tf.transpose here an not K.transpose since the latter does not allow perm argument
    invcov = tf.transpose(invcov, perm=[0,2,1])     # invcov shape is now (batch, 5,5)

    # Difference between prediction and true state vectors
    y_diff = y_pred - y_state

    # n.b. use "batch_dot" and not "dot"!
    y_dot = K.batch_dot(invcov, y_diff)           # y_dot shape is (batch,5,1)
    y_dot = K.reshape(y_dot, (batch_size, 1, 5))  # y_dot shape is now (batch,1,5)
    y_loss = K.batch_dot(y_dot, y_diff)           # y_loss shape is (batch,1,1)
    y_loss = K.reshape(y_loss, (batch_size,))     # y_loss shape is now (batch)
    return y_loss
```

Jefferson Lab

EPSCI