High Level Introduction
Neural Networks

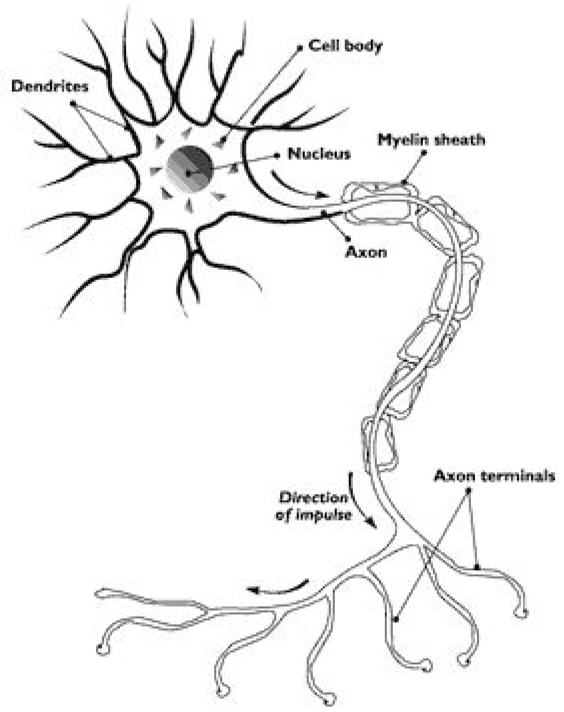
Professor Marie Roch

Readings are listed in the schedule

The goal of this unit is to quickly advance you to the point that you can create a network.
Detailed theory will be taught later.

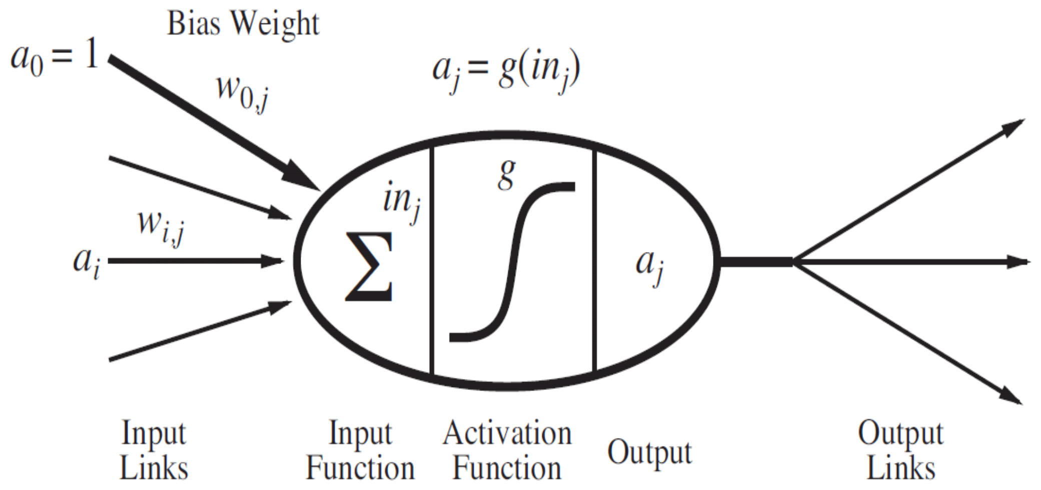
SAN DIEGO STATE UNIVERSITY

Unit – Basic building block



Neuron

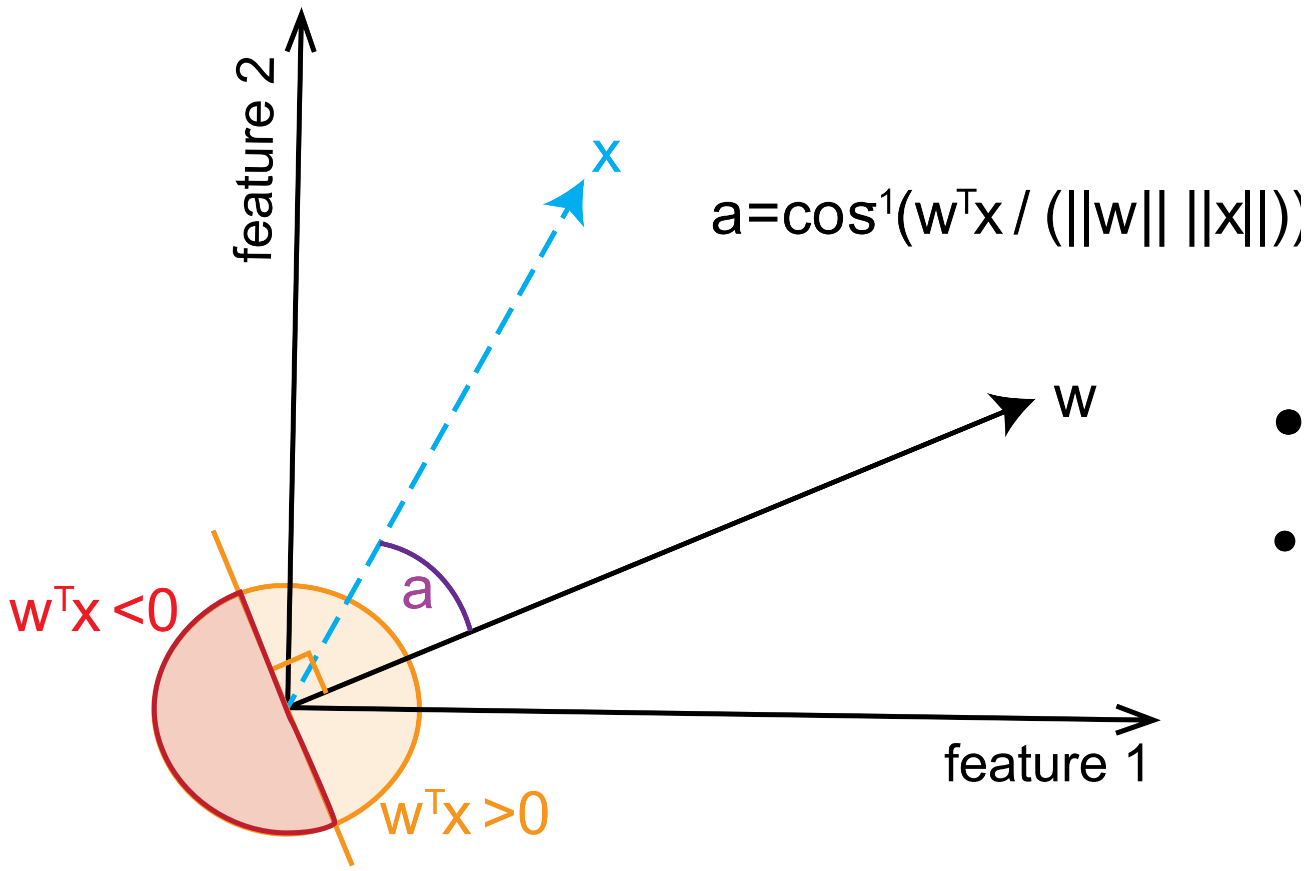
National Institute on Drug Abuse



Loosely inspired model of a neuron

Fig. 18.19 Russell and Norvig

Interpreting weight vectors

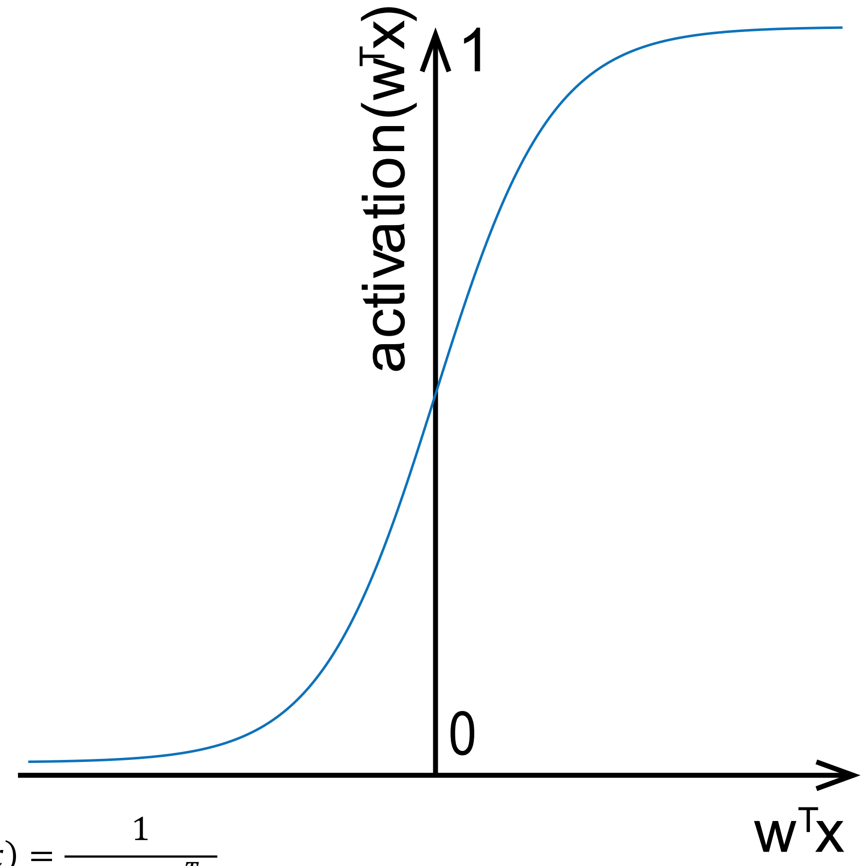


Roch et al. 2021, *Acoustics Today*

- $w^T x \propto \angle a$
- Sign indicates which side of line ⊥ to w vector x falls on

Activation function

- The dot product is passed through an activation function.
- Key ideas about activation functions:
  - nonlinear
  - differentiable
- Common functions:
  - sigmoid (shown)
  - rectified linear unit



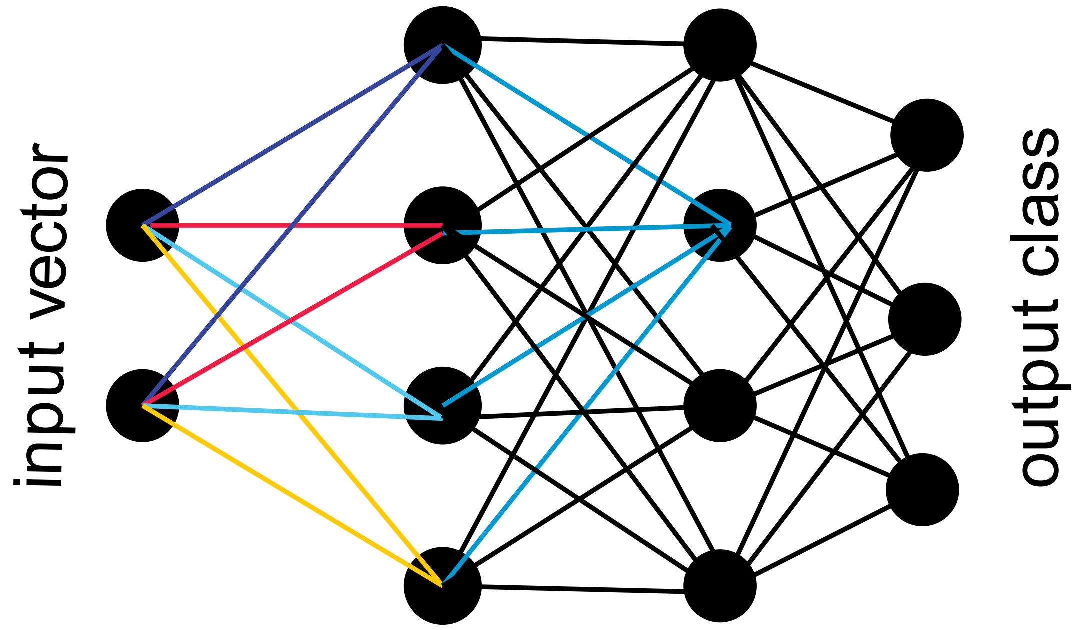
$$\sigma(w^T x) = \frac{1}{1 + e^{-w^T x}}$$

Roch et al. 2021, *Acoustics Today*

SAN DIEGO STATE UNIVERSITY

Putting it together

- Feature vectors are presented to each node of the network
- Each node computes an output
- Subsequent nodes take previous inputs



derived from Roch et al. 2021, *Acoustics Today*

Output layer

- Output neurons encode information
- Common strategy for classification: “one hot”
  - each class given an ouput
  - correct class is 1, others 0

○ 0

● 1

○ 0

e.g. speaker identification:
“Yo Adrian”

San Diego State University

Learning in a neural net

- Learning consists of estimating the weight vectors
- Requires supervised learning
  - Network is trained with example, label
  - We compute the output and determine if it is correct
  - If incorrect, we determine how to adapt the weights to improve them

Learning in a neural net

Broad strokes for now, details later…

- Example x has label $y \in \{0,1\}$
- Neuron computes $\hat{y} = a(w^T x)$ where a is the activation
- Loss: measure of difference between $\hat{y}$ and y
  e.g. squared error: $(\hat{y} - y)^2$

What is the loss if we predict correctly?

A common loss metric for categorical data is cross entropy (covered later)

Learning in a neural net

- Weights are updated based on training examples, labels, and predictions
- The derivative of the loss tells us which way increases the loss the fastest
- Moving in the opposite direction decreases the loss
- Computed on each element of weight vector

$$w_i = w_i - \alpha \frac{\partial}{\partial w_i} Loss(w)$$

$\alpha \triangleq$ learning rate

Learning in a neural net

- This only handles the last node
- We can distribute portions of the loss to the nodes that feed into the last nodes
- *Backpropagation* algorithm lets us determine loss at nodes other than at the output and adapt their weights
- Each pass through the training data is called an *epoch*

Update rule

Consider what happens as we change how we estimate the gradient of the loss

- every sample – Changes a lot
- all training data – Very stable

SAN DIEGO STATE UNIVERSITY

Update rule

A good compromise is to update weights based on a small batch of samples

- Provides more stable gradient
- Faster than updating once per epoch

We typically iterate through the data until the network converges

Network design

- Adding layers or adding width to a layer enables learning more complicated functions
- It also enables overfitting (learning training examples too well)

- Regularization strategies help prevent this

L2 regularization

- Places constraints on the weights
- We still try to minimize the loss, but we add to the loss function a function of the weights
- In L2 regularization,

$$= Loss(\hat{y}, y) + \alpha_R w^T w$$

- Tends to keep weights smaller. L2 weight, α_R, is typically small, e.g. 0.01.

Recap

- Weights carry the “information” of a neural network
- Activation functions provide the ability to solve nonlinear problems
- Loss functions measure whether are network predicts data correctly
- Learning minimizes loss using backpropagation
- Regularization helps to prevent overfitting

Modern Neural Net Libraries

- Specify computational graphs
- Perform automatic derivative computation, making gradient estimation trivial
- Support a wide variety of unit types and the ability to create custom ones
- Examples: pytorch and Tensorflow

Keras
κέρας

- Library designed to simplify neural net specification
- Originally designed to work with several neural net packages including Tensorflow
- Now part of the official Tensorflow distribution
- Keras 3.0 (fall 2023) introduces backend support for Pytorch and JAX

http://vem.quantumunlimited.org/the-gates-of-horn/

SAN DIEGO STATE UNIVERSITY

Keras

- Advantages
  - High-level specification of neural nets and other computation.
  - Transparent GPU vs non-GPU programming
  - Rapid specification

Keras concepts : Models

Models can be:

- Specified: Functionality is specified by invoking model methods, e.g. add a new layer of N nodes.
- Compiled: A compile method writes the back-end code to generate the model
- Fitted: Optimization step where weights are learned
- Evaluated: Tested on new data

Keras concepts : Models

We can use a Sequential model for a feed-forward network

```
from tensorflow.keras.models import Sequential
model = Sequential()
```

Keras concepts: Layers

- Layers can be added to a model
- Dense layers
  - compute $f(W^T x+b)$
  - user specifies
    - number of units
    - input/output tensor shapes
      (tensors are N-dimensional arrays)
    - activation functions
    - other options…

A Keras model

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input

model = Sequential()

# Three category prediction with 2 hidden layers
# and 30 features, categorical output (3 categories)
model.add(Input(shape=(30,)))   # Note (30,) is a tuple w/one element
model.add(Dense(10, activation='relu'))
model.add(Dense(10, activation='relu'))
# Output probability of each category
model.add(Dense(3, activation='softmax'))
```

code for Tensorflow 2+

```
# Create the computational graph
# Specify type of gradient descent, loss metric, and
# measurement metric
model.compile(optimizer = "Adam",
              loss = "categorical_crossentropy",
              metrics = ["accuracy"])

# Not needed: prints architecture summary
model.summary()

# We need examples and labels for supervised learning
# examples:  samples X features numpy.array
examples = get_features()  # you write this

# samples X 1 vector of our 3 categories
labels = get_labels()  # you write this
```

```
from tensorflow.keras.utils import to_categorical

# Our network uses a Multinoulli distribution to
# output one of three choices.  Our labels are scalars,
# we need to convert these to vectors:
# 0 -> [1 0 0], 1 -> [0 1 0], 2 -> [0 0 1]
# this is sometimes called a "one-hot" vector

onehotlabels = to_categorical(labels)

# train the model
# 10 passes (epochs) over data, mini-batch size 100
model.fit(examples, labels, batch_size=100, epochs=10)
```

Using a trained model

- To predict outputs

```
results = model.predict(examples)
```

  - results is Nx3 probabilities
  - What are the following?
    - `np.sum(results, axis=1)`
    - `np.argmax(results, axis=1)`

Using a trained model

- To evaluate performance

```
# Returns list of metrics
results = model.evaluate(test_examples, test_labels)

# model.metrics_names tells us what was measured
# here:  ['loss', 'categorical_accuracy’]

print(results[1])   # accuracy
# In some fields, it is common to report error: 1 - accuracy
```

To install SciKit: conda install scikit-learn

N-fold cross validation

```
# Create a plan for k-fold testing with shuffling
# of examples using SciKit's KFold cross validation
from sklearn.model_selection import KFold

# Randmize examples within each fold
kfold = KFold(n_folds, shuffle=True)

# Generate indices that can be used to split into training
# and test data, e.g. examples[train_idx]
for (train_idx, test_idx) in kfold.split(Examples, Labels):
    # normally, we would gather results about each fold
    train_and_evaluate(examples, one_hot_labels,
                       train_idx, test_idx)
```

SAN DIEGO STATE UNIVERSITY

An architecture for building networks

- Data-driven network construction
- Store constructors and their arguments in a list of tuples e.g.

```
network = [
    (Dense, [in_N], {‘activation’: ‘relu’})
    (Dense, [out_N], {‘activation’: ‘softmax’})]
```

- Tuple:
  – layer name
  – list of positional arguments
  – dictionary of named arguments

An architecture for building networks

Model construction is easy:

- Create a sequential model
- Loop over tuples
  - – Call the layer type to construct a layer
    - Use * to pass in positional args: *tuple[1]
    - Use ** to treat dictionary as named args: **tuple[2]
  - – Add the layer to the model

```
# How I build feed forward models (assumes appropriate  – Marie Roch

# model specification (generic)
modelgen = lambda input_nodes, layer_width, outputN :
   [(Dense, [layer_width], {'activation':'relu', 'input_dim':input_nodes}),
    (Dense, [layer_width], {'activation':'relu', 'input_dim':layer_width}),
    (Dense, [outputN], {'activation':'softmax', 'input_dim':layer_width})
   ]

# Now  we can instantiate the list with a given number of features and output.
# (These would all be calculated)
dim = 20   # Example dimension space
width  =  50 # Number of neurons per layer
classesN = 3

# Build a new list with specific parameters
design = modelgen(dim, width, classesN)

model = build_model(design)  # Generate network computation graph
model.compile()
```

```
from tensorflow.keras.models import Sequential
import tensorflow.keras.backend as K

def build_model(specification, name="model"):
    """build_model - specification list
    Create a model given a specification list
    Each element of the list represents a layer and is formed by a tuple.

    (layer_constructor,
     positional_parameter_list,
     keyword_parameter_dictionary)

    Example, create M dimensional input to a 3 layer network with
    20 unit ReLU hidden layers and 5 category softmax output layer

    [(Dense, [20], {'activation’: 'relu'}),
     (Dense, [20], {'activation’: 'relu'}),
     (Dense, [5], {'activation':'softmax'})
    ]

    Wrappers are supported by creating a 4th item in the tuple/list
    that consists of a tuple with 3 items:
       (WrapperType, [positional args], {dictionary of arguments})

    The WrapperType is wrapped around the specified layer which is assumed
    to be the first argument of the constructor.  Additional positional
    argument are taken from the second item of the tuple and will *follow*
    the wrapped layer argument.  Dictionary arguments
    are applied as keywords.

     For example:
    (Dense, [20], {'activation':'relu'}, (TimeDistributed, [], {}))

    would be equivalent to calling TimeDistributed(Dense(20, activation='relu'))
    If TimeDistributed had positional or named arguments, they would be placed
    inside the [] and {} respectively.  Remember that the wrapped layer (Dense)
    in this case is *always* the first argument to the wrapper constructor.

    Author - Marie A. Roch, 12/2017, updated for tensorflow 2 in 2020
    """

    K.name_scope(name)
    model = Sequential()
```

SAN DIEGO STATE UNIVERSITY

```
for item in specification:
   layertype = item[0]
   # Construct layer and add to model
   # This uses Python's *args and **kwargs constructs
   #
   # In a function call, *args passes each item of a list to
   # the function as a positional parameter
   #
   # **args passes each item of a dictionary as a keyword argument
   # use the dictionary key as the argument name and the dictionary
   # value as the parameter value
   #
   # Note that *args and **args can be used in function declarations
   # to accept variable length arguments.
   layer = layertype(*item[1], **item[2])

   if len(item) > 3:
      # User specified wrapper
      wrapspec = item[3]
      # Get type, positional args and named args
      wraptype, wrapposn, wrapnamed = wrapspec
      wlayer  = wraptype(layer, *wrapposn, **wrapnamed)
      model.add(wlayer)
   else:
      # No wrapper, just add it.
      model.add(layer)

return model
```

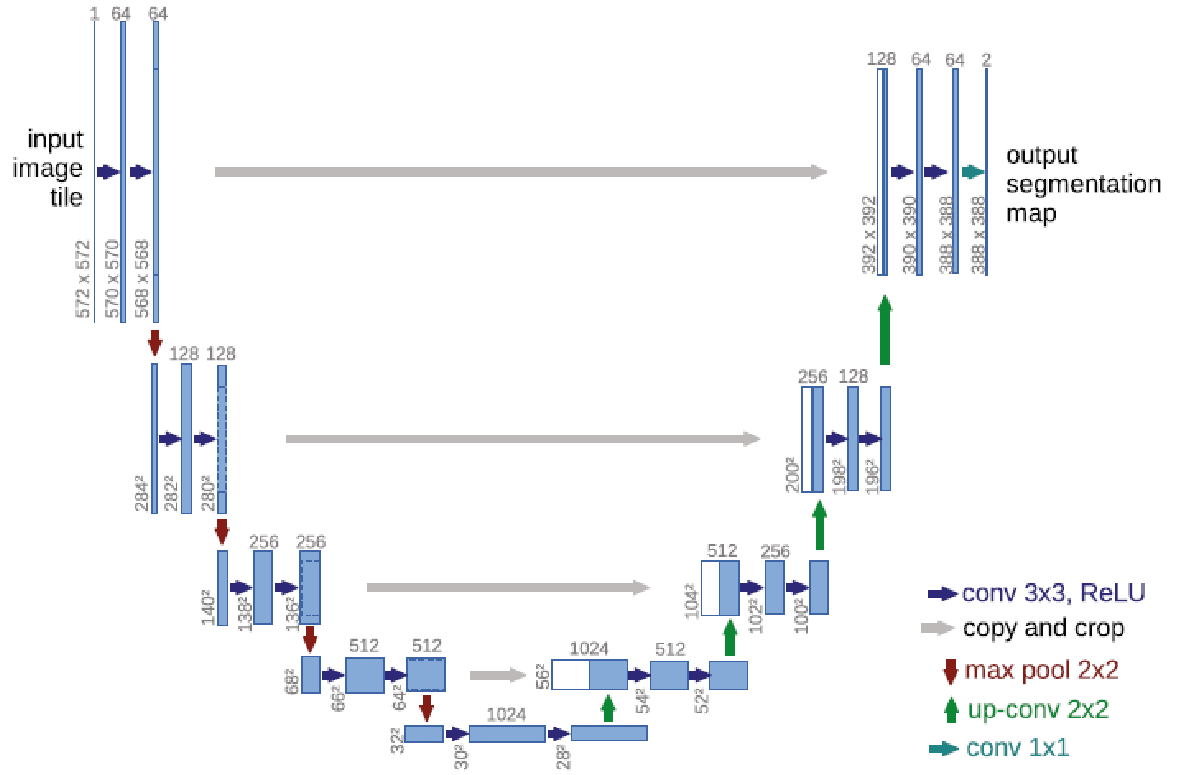
Specifying networks as data structures

```
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.regularizers import l2

arch_abstract = lambda indim, layer_width, penalty, outdim : [
   (Input, [], {'shape':indim}),
   (Dense, [layer_width], {'activation':'relu', 'kernel_regularizer':l2(penalty)}),
   (Dense, [outdim], {'activation':'softmax'})
      ]

# instantiate the network with an input shape of (10,), 25 neurons,
# an L2 penalty of 0.01 and 5 output categories
arch_actual = arch_abstract((10,), 25, 0.01, 5)
model = build_model(arch_actual)
```

San Diego State University

Other types of models

- Not all models are sequential:



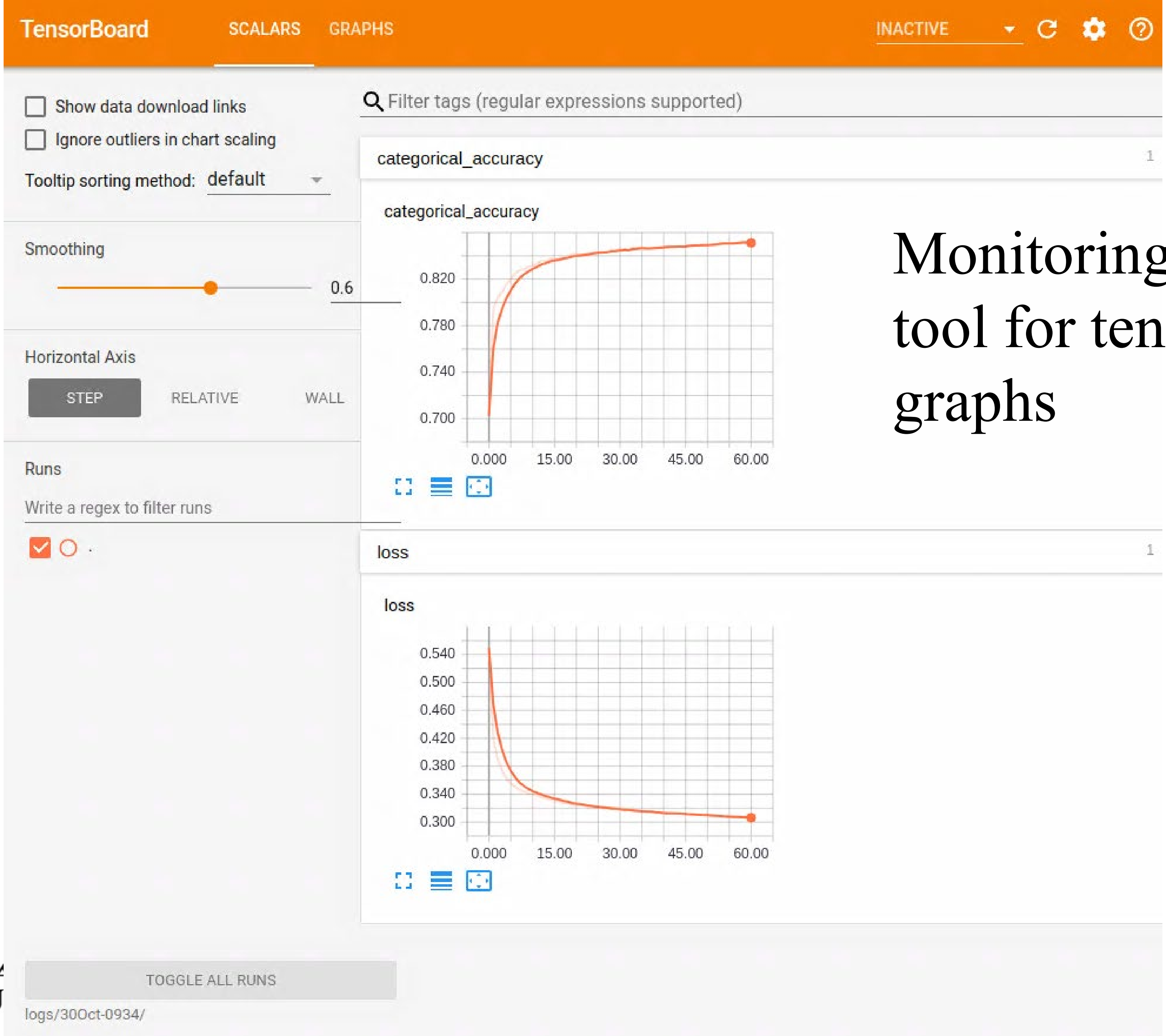
Ronneberger et al.'s U-Net MICCAI 2015

SAN DIEGO STATE UNIVERSITY

Complicated architectures

Use the functional API

```
# This returns a tensor
inputs = Input(shape=(N_inputs,))
# a layer instance is callable on a tensor, and returns a tensor
x = Dense(N_width, activation='relu')(inputs)
x = Dense(N_width, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)
# This creates a model that includes the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop', loss='categorical_crossentropy’,
                        metrics=['accuracy’])
model.fit(data, labels) # starts training
```


TensorBoard SCALARS GRAPHS INACTIVE

Show data download links

Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0.6

Horizontal Axis: STEP RELATIVE WALL

Runs

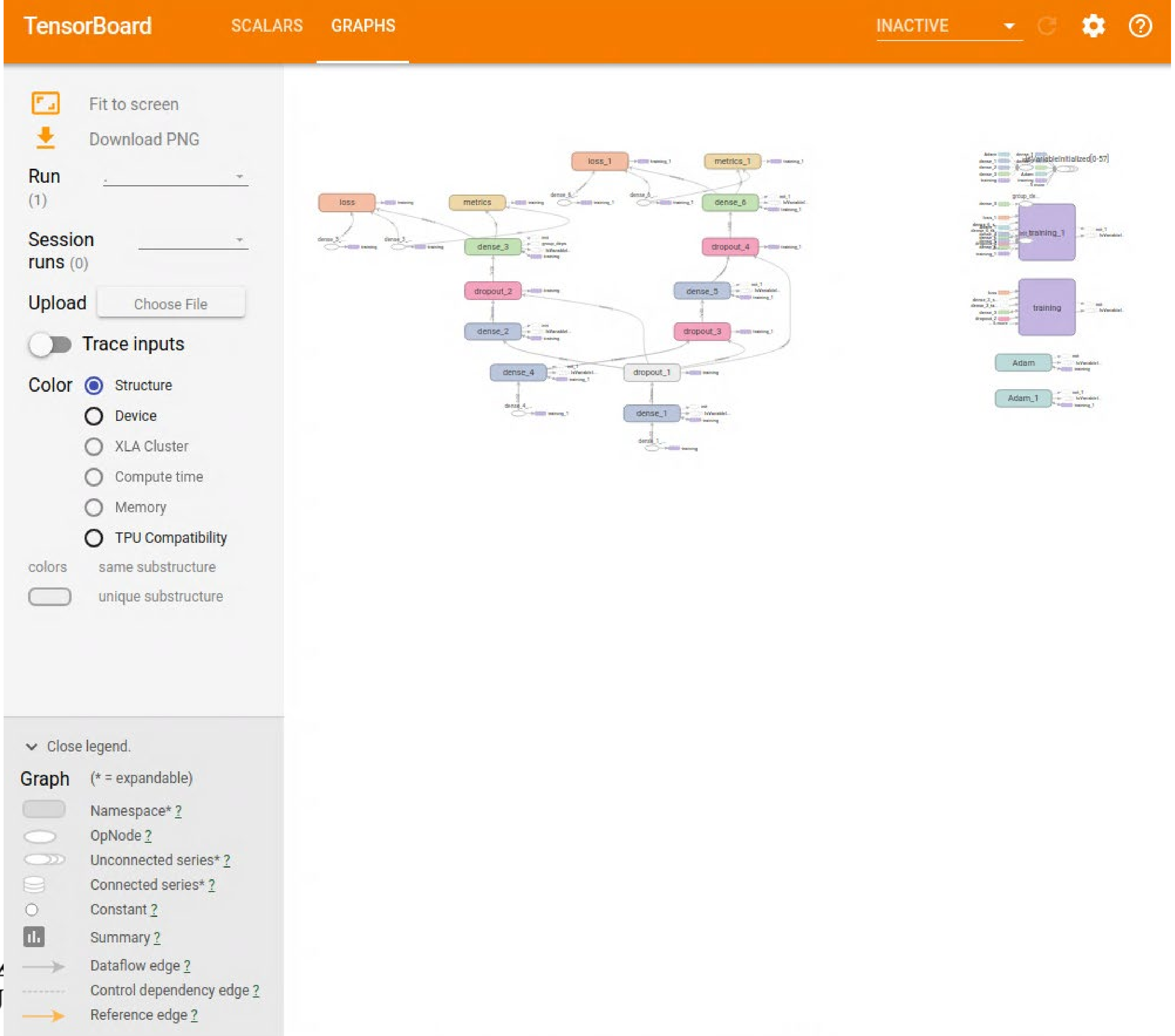
Write a regex to filter runs

TOGGLE ALL RUNS

logs/30Oct-0934/

Filter tags (regular expressions supported)

Monitoring tool for tensor graphs


TensorBoard
SCALARS
GRAPHS
INACTIVE
Fit to screen
Download PNG
Run (1)
Session runs (0)
Upload
Choose File
Trace inputs
Color
Structure
Device
XLA Cluster
Compute time
Memory
TPU Compatibility
colors
same substructure
unique substructure
Close legend.
Graph (* = expandable)
Namespace* ?
OpNode ?
Unconnected series* ?
Connected series* ?
Constant ?
Summary ?
Dataflow edge ?
Control dependency edge ?
Reference edge ?
loss
metrics
loss_1
metrics_1
dense_3
dense_6
dropout_2
dropout_4
dense_2
dense_5
dropout_3
dense_4
dropout_1
dense_1
training_1
training
Adam
Adam_1

TensorBoard

```
from tensorflow.keras.callbacks import TensorBoard
…
tensorboard = TensorBoard(
         # Write to logs directory, e.g. logs/30Oct-05:00
        log_dir="logs/{}".format(time.strftime('%d%b-%H%M')),
        histogram_freq=0,
        write_graph=True,  # Show the network
)
    # train the net
    model.fit(examples, onehotlabels,
              epochs=epochs, callbacks=[loss, tensorboard])
```

Then start tensorboard from the command prompt:

```
➢  tensorboard –logdir logs/30Oct-05:00
TensorBoard 1.5.1 at http://localhost:6006 (Press CTRL+C to
quit)
```

Point chrome at the URL and off you go…

There are lots of tutorials if you want to use advanced features.