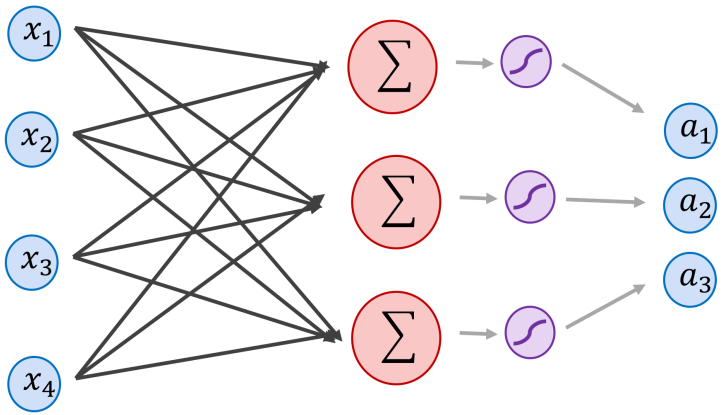
11-695: AI Engineering
Customizing NNs III

LTI/SCS

Spring 2020

Outline

Non-Linear Activations



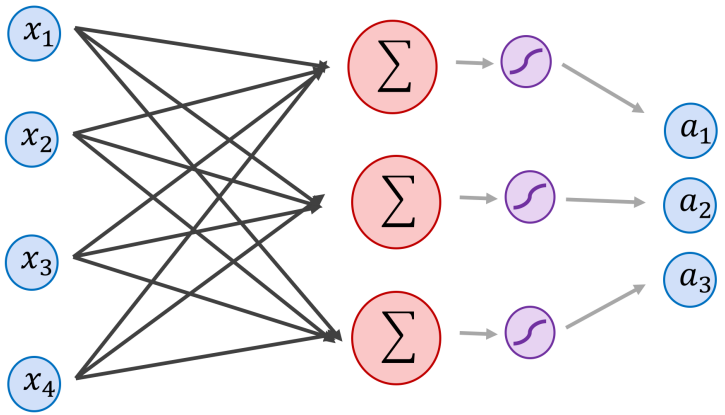
- Recall: every NN has this form of operations

$$\hat{\mathbf{y}} = (\phi_n \circ \mathbf{f}_{W_n} \circ \phi_{n-1} \circ \mathbf{f}_{W_{n-1}} \ldots \phi_1 \circ \mathbf{f}_{W_1})(\mathbf{X})$$

- Two basic operations
  - Linear: $o_i = \mathbf{f}_{W_i}(a_{i-1}) = {W_i}^T a_{i-1} + b_i$
  - Nonlinear (by activation functions): $a_i = \phi(o_i)$

Image credit: Vicente Ordóñez Román

Non-Linear Activations



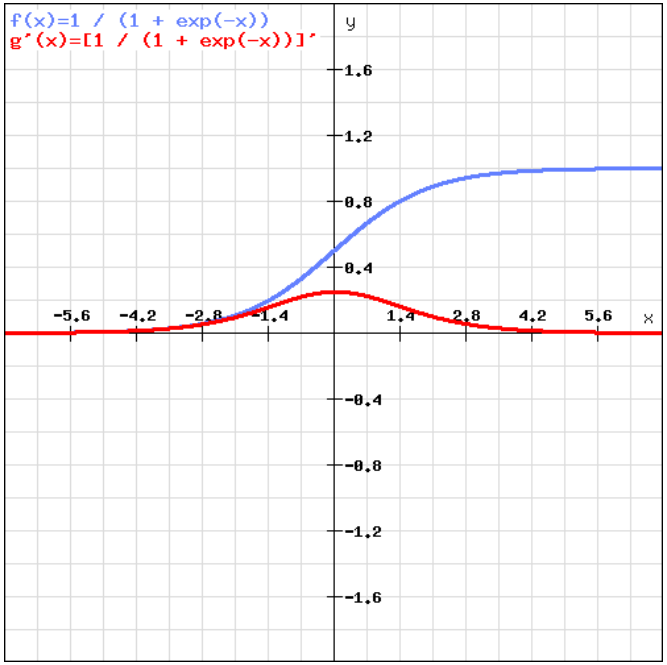
- Recall: every NN has this form of operations

$$\hat{\mathbf{y}} = (\phi_n \circ \mathbf{f}_{W_n} \circ \phi_{n-1} \circ \mathbf{f}_{W_{n-1}} \ldots \phi_1 \circ \mathbf{f}_{W_1})(\mathbf{X})$$

- Two basic operations
  - Linear: $o_i = \mathbf{f}_{W_i}(a_{i-1}) = {W_i}^T a_{i-1} + b_i$
  - Nonlinear (by activation functions): $a_i = \phi(o_i)$
- To model non linearity, activations need be *non-linear*.

Image credit: Vicente Ordóñez Román

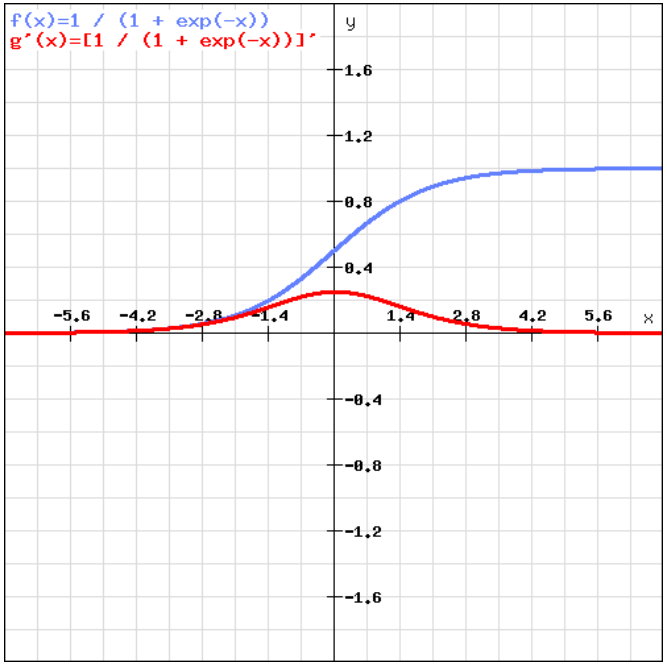
(Logistic) Sigmoid



- $\sigma(x) = 1/(1 + \exp(-x))$ and $\sigma'(x) = \sigma(x)(1 - \sigma(x))$

Plot tool credit: `rechneronline.de`

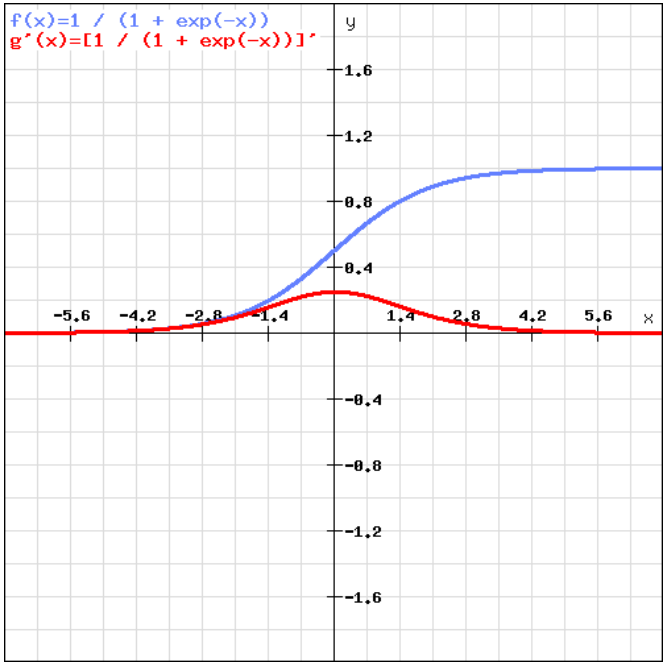
(Logistic) Sigmoid



- $\sigma(x) = 1/(1 + \exp(-x))$ and $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- Both are always positive

Plot tool credit: `rechneronline.de`

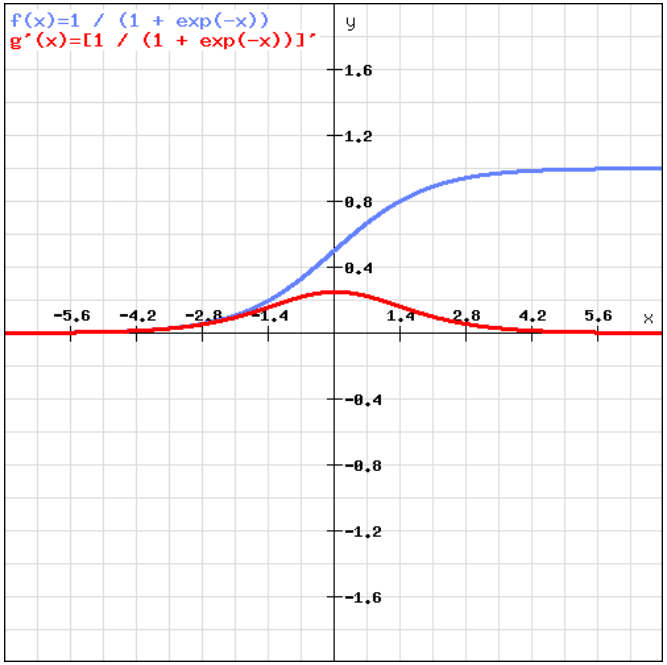
(Logistic) Sigmoid



- $\sigma(x) = 1/(1 + \exp(-x))$ and $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- Both are always positive
- Squash unbounded values to $[0, 1]$, popular as logistic function

Plot tool credit: `rechneronline.de`

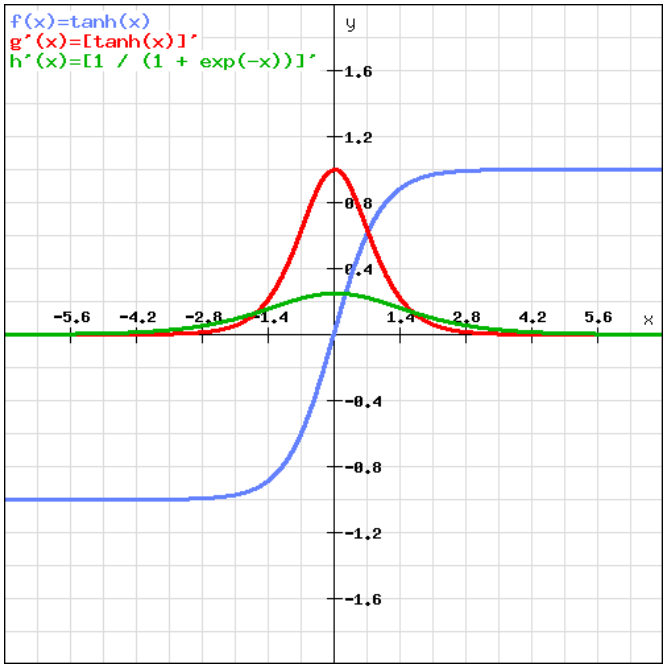
(Logistic) Sigmoid



- $\sigma(x) = 1/(1 + \exp(-x))$ and $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- Both are always positive
- Squash unbounded values to $[0, 1]$, popular as logistic function
- Gradients vanishing problem

Plot tool credit: `rechneronline.de`

Tanh (Hyperbolic Tangent)

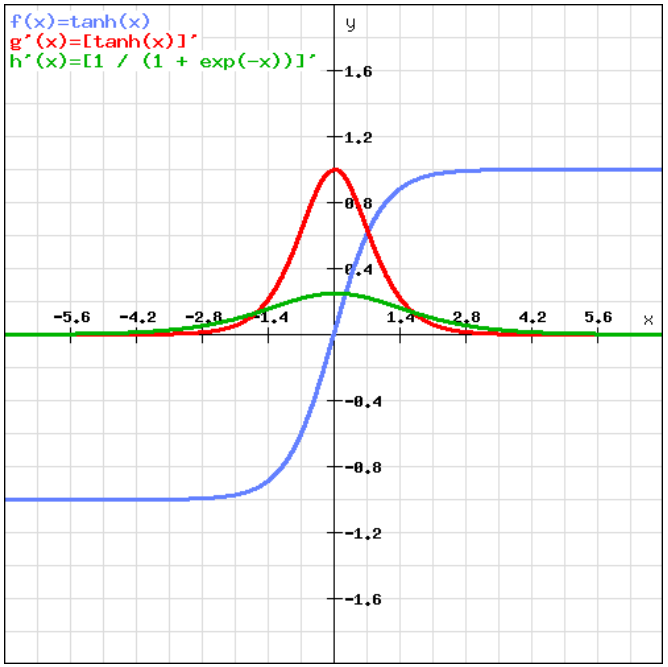


- $f(x) = \tanh(x)$ and $f'(x) = 1 - f^2(x)$

[1] http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

Plot tool credit: rechneronline.de

Tanh (Hyperbolic Tangent)

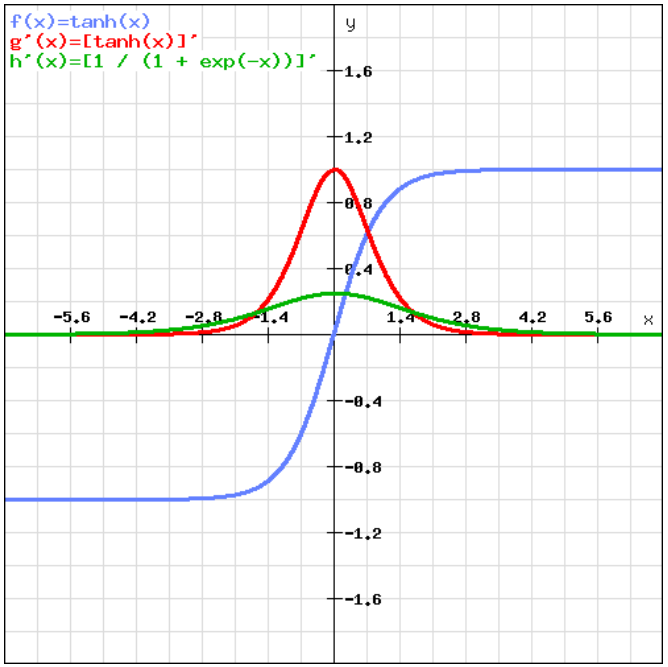


- $f(x) = \tanh(x)$ and $f'(x) = 1 - f^2(x)$
- Squash unbounded values to $[-1, 1]$, also used as logistic function

[1] http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

Plot tool credit: rechneronline.de

Tanh (Hyperbolic Tangent)

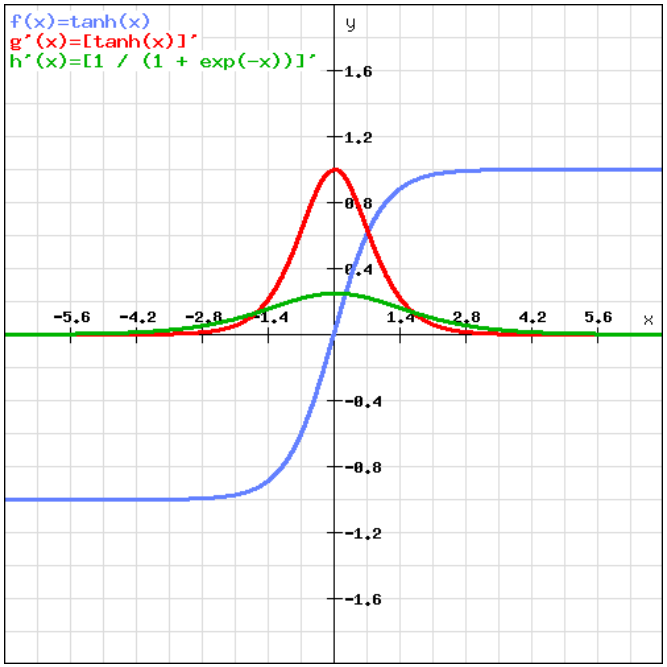


- $f(x) = \tanh(x)$ and $f'(x) = 1 - f^2(x)$
- Squash unbounded values to $[-1, 1]$, also used as logistic function
- Symmetric around origin, preferred over sigmoid[1]

[1] http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

Plot tool credit: rechneronline.de

Tanh (Hyperbolic Tangent)

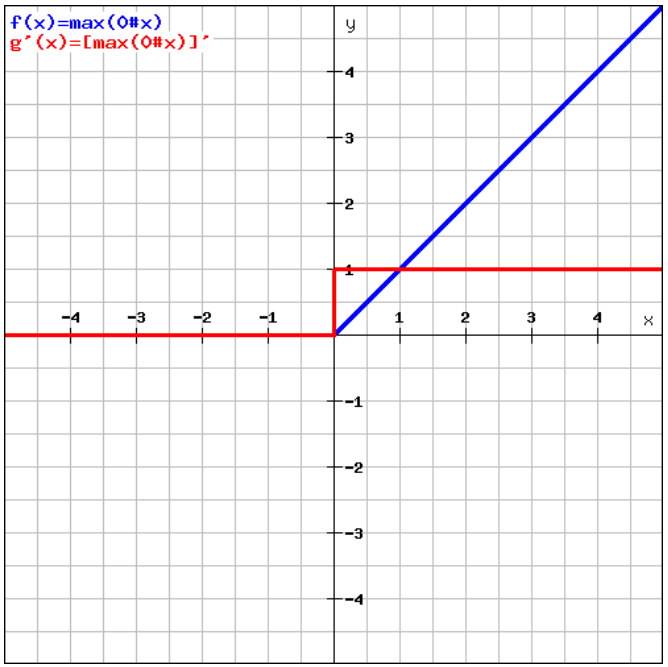


- $f(x) = \tanh(x)$ and $f'(x) = 1 - f^2(x)$
- Squash unbounded values to $[-1, 1]$, also used as logistic function
- Symmetric around origin, preferred over sigmoid[1]
- Also, gradients vanishing problem

[1] http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

Plot tool credit: rechneronline.de

ReLU (Rectified Linear Unit)

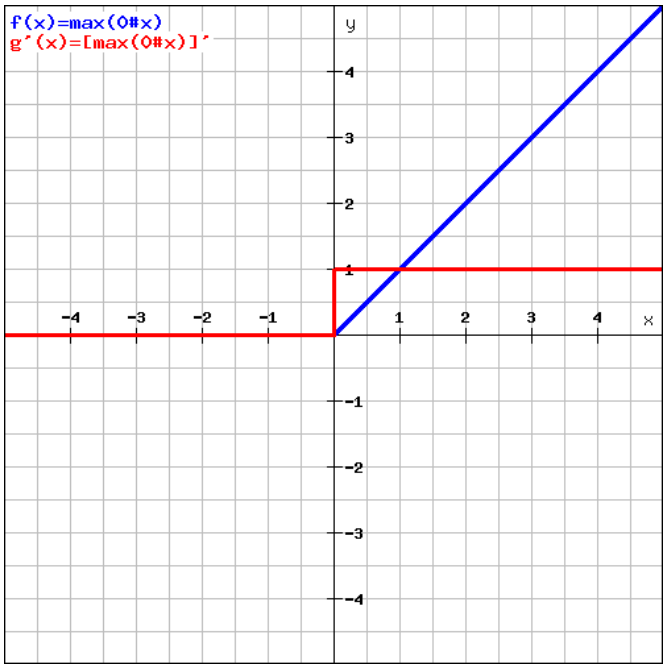


- $f(x) = x^+ = \max(0, x)$ and $f'(x) = \mathbf{1}_{x \geq 0}(x)$

Plot tool credit: `rechneronline.de`

ReLU (Rectified Linear Unit)

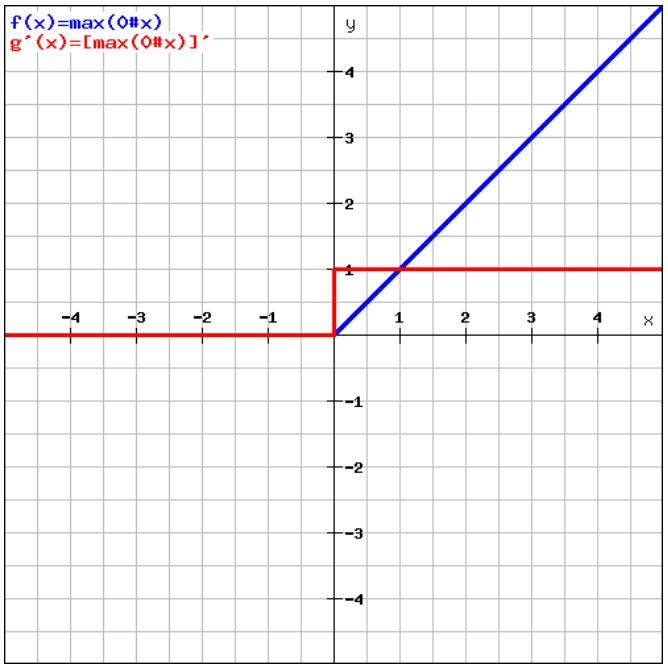
Carnegie Mellon



- $f(x) = x^+ = \max(0, x)$ and $f'(x) = \mathbf{1}_{x \geq 0}(x)$
- Both are *monotonic*, $f(x)$ is half-unbounded

Plot tool credit: `rechneronline.de`

ReLU (Rectified Linear Unit)

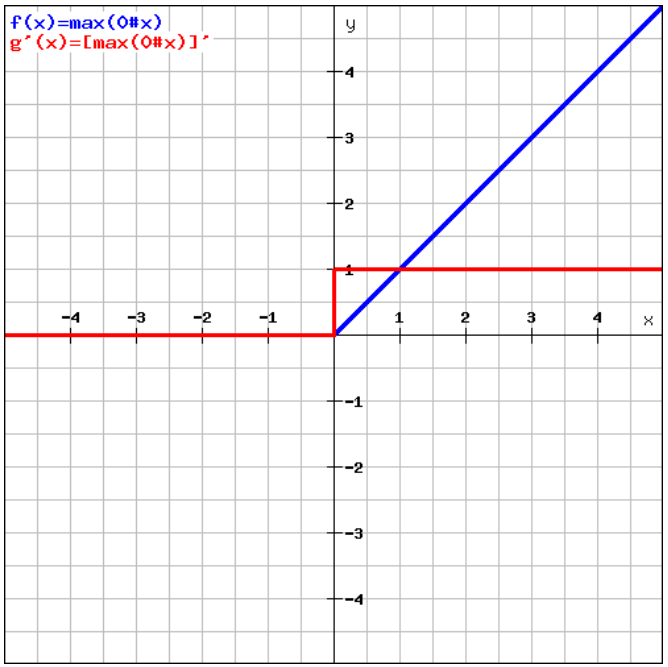


- $f(x) = x^+ = \max(0, x)$ and $f'(x) = \mathbf{1}_{x \geq 0}(x)$
- Both are *monotonic*, $f(x)$ is half-unbounded
- Inexpensive computation

Plot tool credit: `rechneronline.de`

ReLU (Rectified Linear Unit)

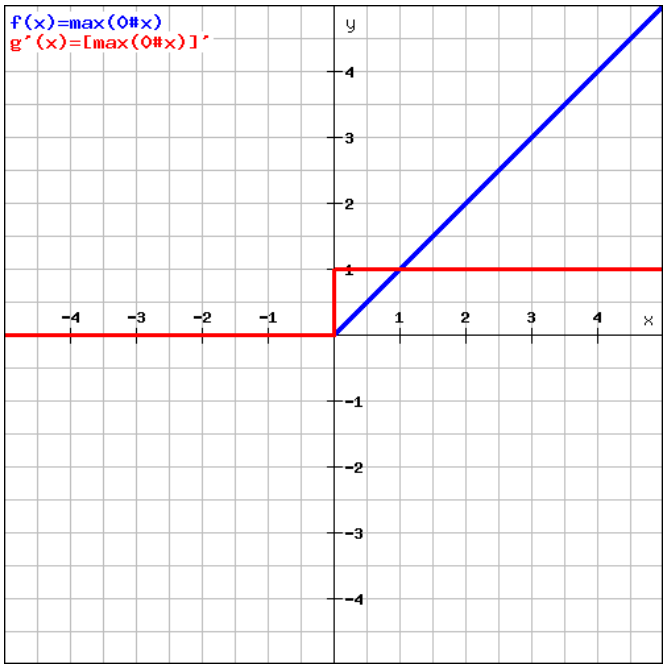
Carnegie Mellon



- $f(x) = x^+ = \max(0, x)$ and $f'(x) = \mathbf{1}_{x \geq 0}(x)$
- Both are *monotonic*, $f(x)$ is half-unbounded
- Inexpensive computation
- Avoid activating *all* neurons: sparse representations

Plot tool credit: `rechneronline.de`

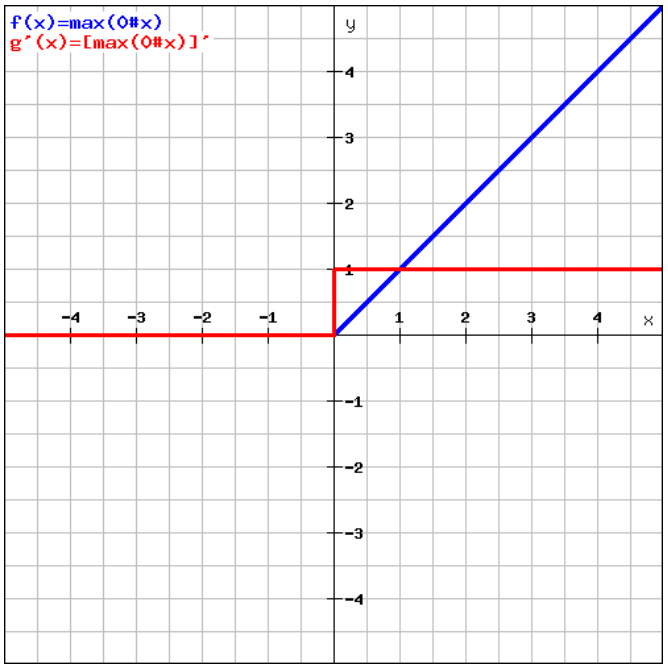
ReLU (Rectified Linear Unit)



- $f(x) = x^+ = \max(0, x)$ and $f'(x) = \mathbf{1}_{x \geq 0}(x)$
- Both are *monotonic*, $f(x)$ is half-unbounded
- Inexpensive computation
- Avoid activating *all* neurons: sparse representations
- *Hate* negative values,

Plot tool credit: `rechneronline.de`

ReLU (Rectified Linear Unit)

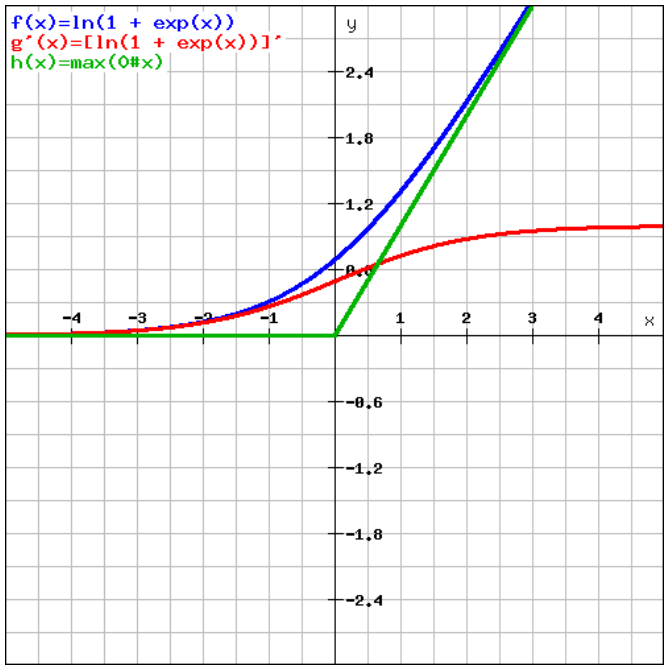


- $f(x) = x^+ = \max(0, x)$ and $f'(x) = \mathbf{1}_{x \geq 0}(x)$
- Both are *monotonic*, $f(x)$ is half-unbounded
- Inexpensive computation
- Avoid activating *all* neurons: sparse representations
- *Hate* negative values, but is most used by far

Plot tool credit: `rechneronline.de`

Softplus[2]: an attempt to fix ReLU

Carnegie Mellon



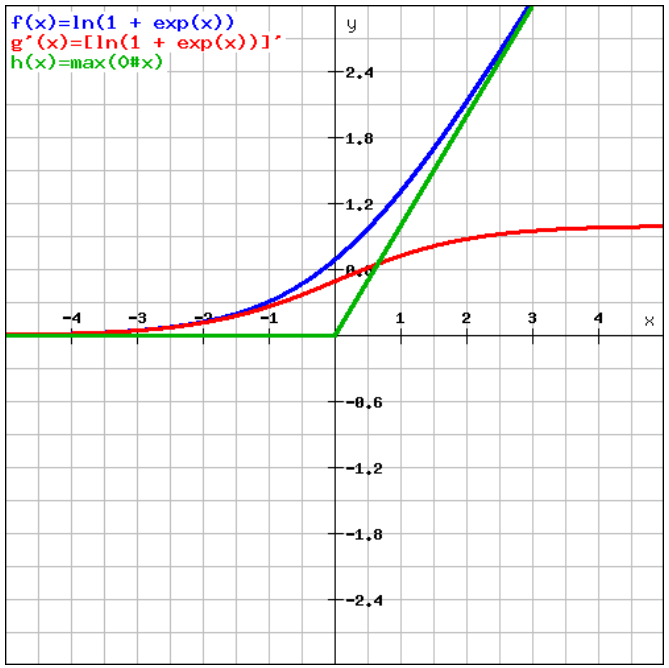
- $f(x) = \ln(1 + \exp(x))$ and $f'(x) = \sigma(x)$

[2] http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf

Plot tool credit: rechneronline.de

Softplus[2]: an attempt to fix ReLU

Carnegie Mellon



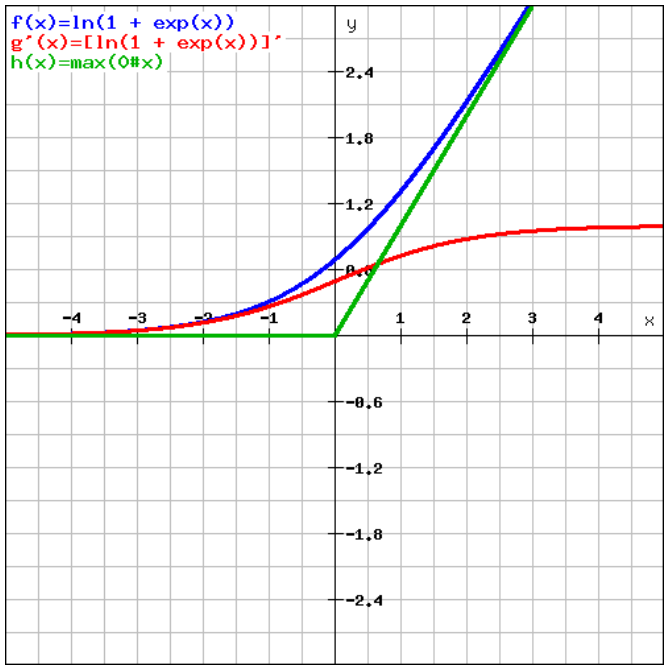
- $f(x) = \ln(1 + \exp(x))$ and $f'(x) = \sigma(x)$
- Smooth approximation of ReLU, *a.k.a* SmoothReLU

[2] http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf

Plot tool credit: rechneronline.de

Softplus[2]: an attempt to fix ReLU

Carnegie Mellon



- $f(x) = \ln(1 + \exp(x))$ and $f'(x) = \sigma(x)$
- Smooth approximation of ReLU, *a.k.a* SmoothReLU
- Avoid zero saturation

[2] http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf

Plot tool credit: rechneronline.de

Softplus: an attempt to fix ReLU

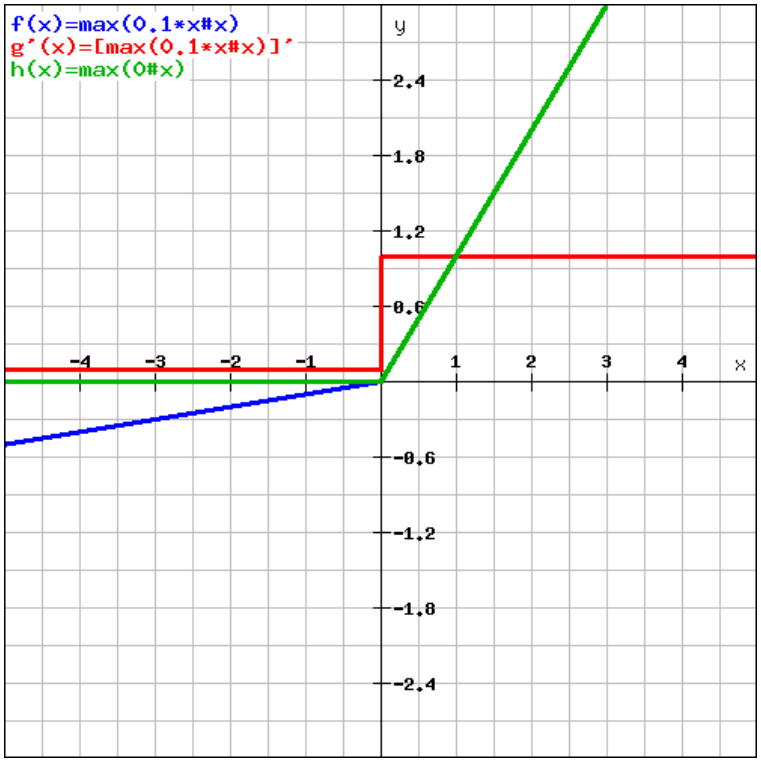
Carnegie Mellon

Neuron	MNIST	CIFAR10	NISTP	NORB
***With* unsupervised pre-training**				
Rectifier	**1.20%**	**49.96%**	**32.86%**	**16.46%**
Tanh	**1.16%**	**50.79%**	35.89%	17.66%
Softplus	**1.17%**	**49.52%**	**33.27%**	19.19%
***Without* unsupervised pre-training**				
Rectifier	**1.43%**	**50.86%**	**32.64%**	**16.40%**
Tanh	1.57%	52.62%	36.46%	19.29%
Softplus	1.77%	53.20%	35.48%	17.68%

- $f(x) = \ln(1 + \exp(x))$ and $f'(x) = \sigma(x)$
- Smooth approximation of ReLU, *a.k.a* SmoothReLU
- Avoid zero saturation
- But is worse, and more expensive

Image credit: Xavier Glorot *et al.*

Leaky ReLU[3]: an(other) attempt

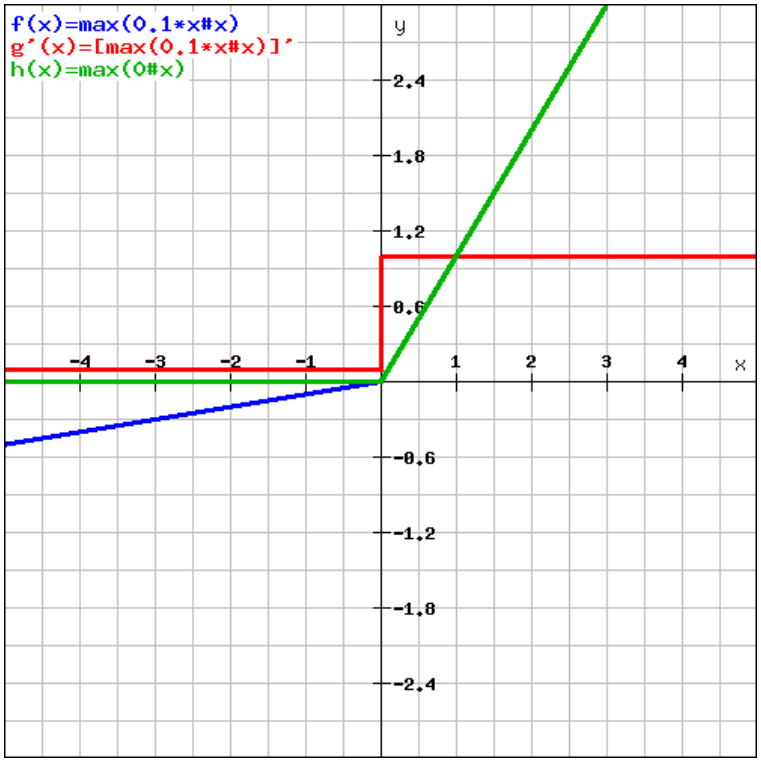


- $f(x) = \max(0.01x, x)$

[3] https://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf

Plot tool credit: rechneronline.de

Leaky ReLU[3]: an(other) attempt

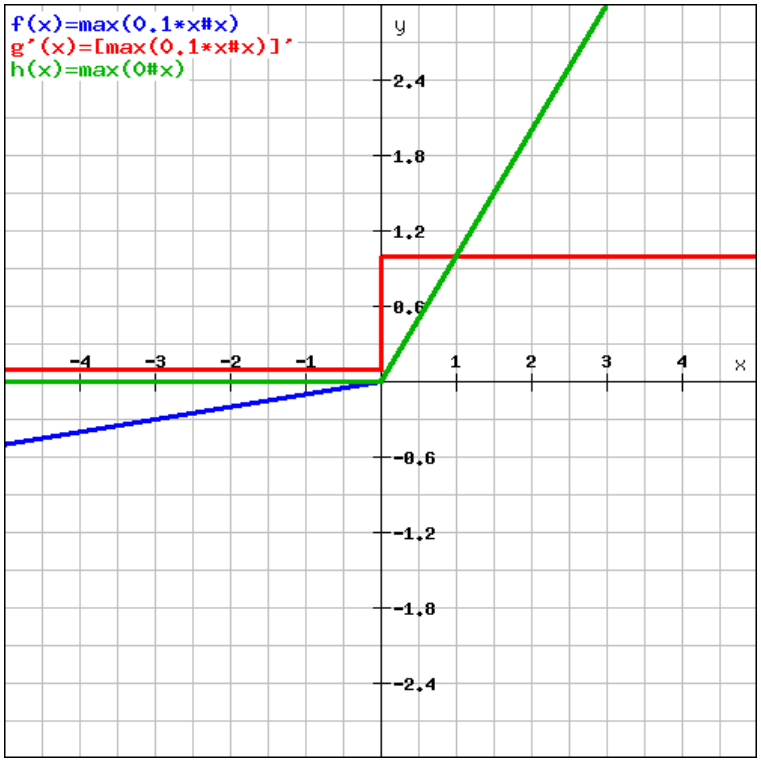


- $f(x) = \max(0.01x, x)$
- Fully unbounded

[3] https://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf

Plot tool credit: rechneronline.de

Leaky ReLU[3]: an(other) attempt



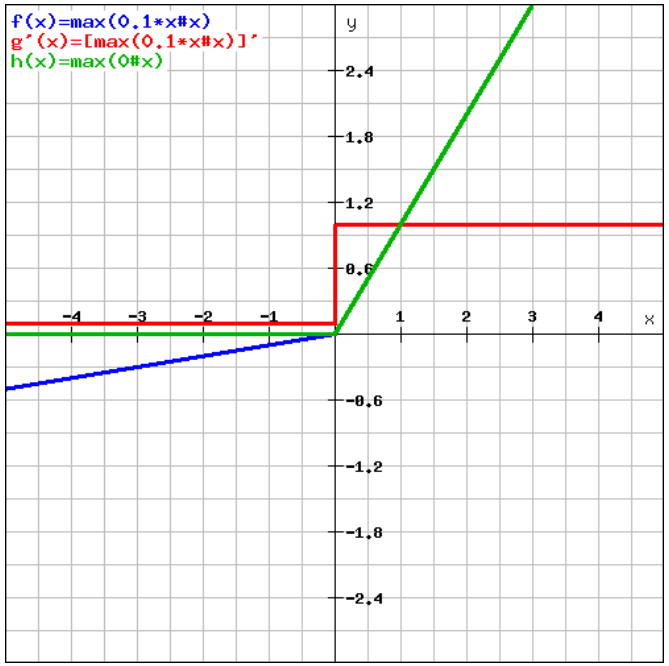
- $f(x) = \max(0.01x, x)$
- Fully unbounded
- Perform "nearly identically to standard rectifier DNNs"

[3] https://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf

Plot tool credit: rechneronline.de

Parametric ReLU[4]: an(other) attempt

Carnegie Mellon

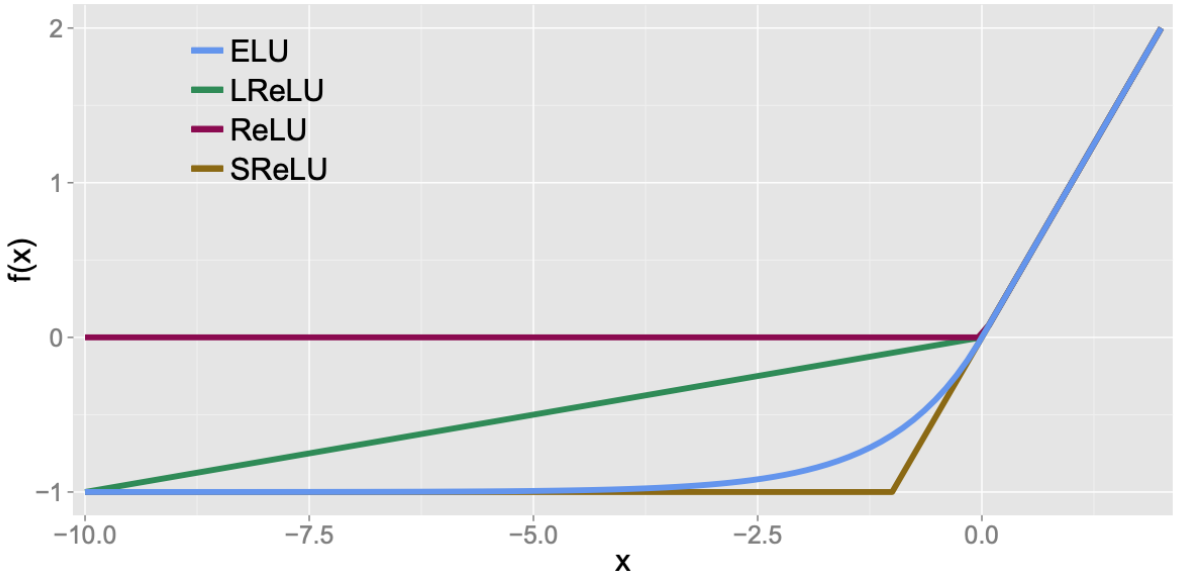


- $f(x) = \max(\alpha x, x)$ for small α
- Also called PReLU
- A generalized version of ReLU

[4] https://arxiv.org/pdf/1502.01852.pdf

Plot tool credit: rechneronline.de

ELU[5]: an(other) attempt



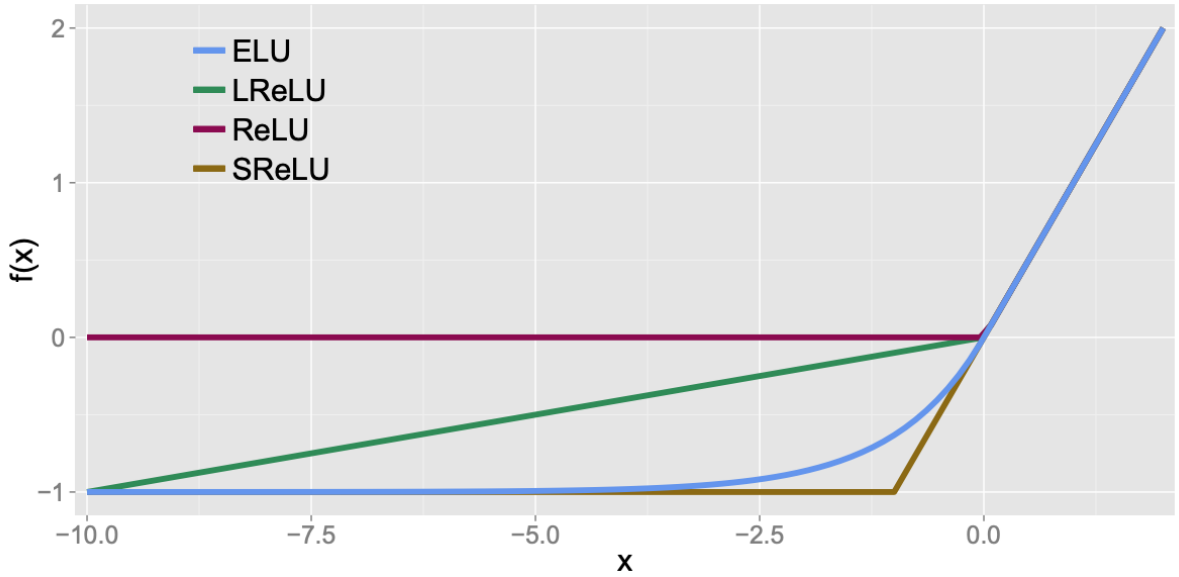
- $f(x) = \alpha(\exp(x) - 1)$ for negative x else x, for small α

[5] https://arxiv.org/pdf/1511.07289.pdf

Image credit: Djork-Arne' Clevert *et al.*

ELU[5]: an(other) attempt

Carnegie Mellon



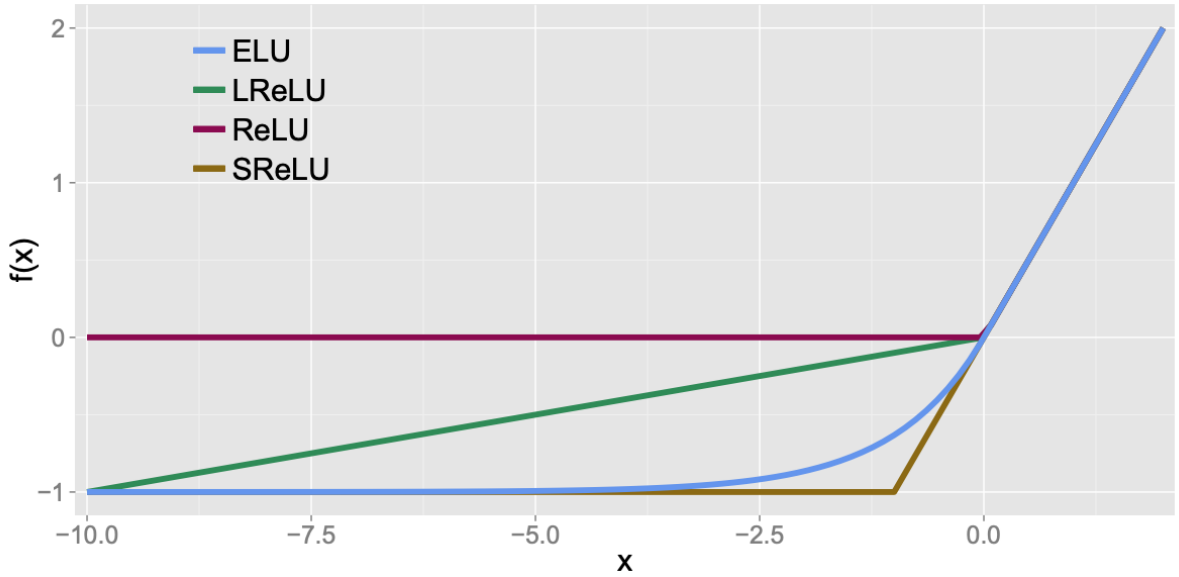
- $f(x) = \alpha(\exp(x) - 1)$ for negative x else x, for small α
- $f'(x) = f(x) + \alpha$ for negative x else 1

[5] https://arxiv.org/pdf/1511.07289.pdf

Image credit: Djork-Arne' Clevert *et al.*

ELU[5]: an(other) attempt

Carnegie Mellon



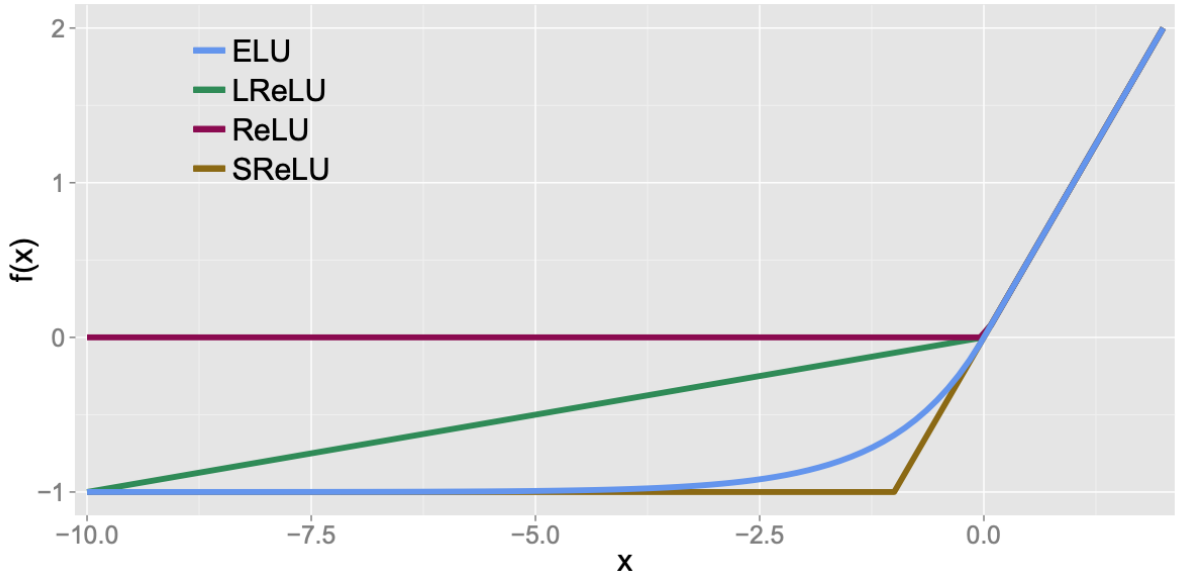
- $f(x) = \alpha(\exp(x) - 1)$ for negative x else x, for small α
- $f'(x) = f(x) + \alpha$ for negative x else 1
- Show better performance than previous rectifiers in many cases

[5] https://arxiv.org/pdf/1511.07289.pdf

Image credit: Djork-Arne' Clevert *et al.*

ELU[5]: an(other) attempt

Carnegie Mellon



- $f(x) = \alpha(\exp(x) - 1)$ for negative x else x, for small α
- $f'(x) = f(x) + \alpha$ for negative x else 1
- Show better performance than previous rectifiers in many cases
- Bring mean of activations to zero thus correct bias shifts

[5] https://arxiv.org/pdf/1511.07289.pdf

Image credit: Djork-Arne' Clevert *et al.*

More activations

Carnegie Mellon

- API: `tf.keras.activations`

Outline

A callback

Carnegie Mellon

`callback` (example from wikipedia)

```
1  def get_square(val):
2      """The callback."""
3      return val ** 2
4
5  def caller(func, val):
6      return func(val)
7
8  caller(get_square, 5) # 25
```

- A function which is passed as a parameter
- Usually work in asynchronous mode
- General usage involves triggering some thing after finishing a job.

A callback in `tf.keras`

Carnegie Mellon

callback_keras

```
1  from tf.keras.callbacks import Callback
2
3  class MyCallback(Callback):
4      ... # more later
5
6  my_callback = MyCallback()
7
8  model.fit(x_train, y_train,
9            batch_size=64,
10           steps_per_epoch=5,
11           epochs=3,
12           verbose=0,
13           callbacks=[my_callback])
```

- Is usually passed through training (`fit` function)

Useful callback: `EarlyStopping`[6]

Carnegie Mellon

early_stopping

```
1  callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
2                                               min_delta=0.001,
3                                               patience=3)
4
5  model.fit(x_train, y_train,
6            batch_size=64,
7            epochs=30,
8            validation_data=(x_val, y_val)),
9            validation_feq=1,
10           callbacks=[callback]) # note: always a list of Callbacks
```

- Usual usage: stop training after some rounds of no improvement
- Usually we monitor `val_loss` or `val_accuracy`

[6] https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping

Useful callback: ModelCheckpoint[7]

Carnegie Mellon

model_checkpoint

```
callbacks = [
    tf.keras.callbacks.ModelCheckpoint(
        filepath='mymodel_{epoch}',
        save_best_only=True,
        monitor='val_loss', # only if this improves
        save_weights_only=True,
        save_freq='epoch',
        verbose=1)
]
model.fit(x_train, y_train,
          batch_size=64,
          epochs=30,
          #validation_data=(x_val, y_val)),
          #validation_feq=1,
          validation_split=0.3,
          callbacks=callbacks)
```

- Usual usage: save after one or a few epochs

[7] https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint

Useful callback: `TensorBoard`[8]

Carnegie Mellon

`tensorboard`

```
callbacks = [
    tf.keras.callbacks.TensorBoard(log_dir='./logs',
                                   histogram_freq=1,
                                   write_graph=True,
                                   write_images=False)
]
model.fit(x_train, y_train,
          batch_size=64,
          epochs=30,
          validation_split=0.3,
          callbacks=callbacks
          )
```

- Usual usage:
  - Monitor training and evaluations
  - Visualize histograms of activations
  - Visualize embeddings
- To visualize: `tensorboard --logdir=./logs`

[8] https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/TensorBoard

Custom Callbacks

Carnegie Mellon

custom_callback

```
class CustomCallback(tf.keras.callbacks.Callback):
  def __init__(self, new_data):
    super(CustomCallback, self).__init__()
    self.data = new_data

  # before/after fit()|eval()|predict()
  def on_(train|test|predict)_(begin|end)(self, logs=None):
    ...
  # before/after each batch processing
  def on_(train|test|predict)_batch_(begin|end)(self, batch, logs=None):
    ...
  # before/after each epoch
  def on_epoch_(begin|end)(self, epoch, logs=None):
    ...
```

- Basically you can customize however you want
- *E.g.* Val/Test after n batches, periodically sample some results to check, ...

Custom Callbacks: `logs` dictionary

Carnegie Mellon

`logs_dict`

```
class SomeCallback(tf.keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []
        self.accuracies = []
    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))
        self.accuracies.append(logs.get('accuracy'))
        print('For batch {}, loss {:7.2f}, acc {:0.2f}.'
            .format(batch, logs['loss'], logs['accuracy']))

model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'], # will go into logs
              callbacks=[SomeCallback()])
# logs dict look like so while on training:
{'batch': 1000, 'size': 32, 'loss': 0.3528049, 'accuracy': 0.8058816}
```

- Store "the loss value, and all the metrics at the end of a batch or epoch"

More about callbacks

Carnegie Mellon

- List of `tf.keras` predefined: ▸ callbacks
- Tutorial: ▸ keras custom callbacks
- Tutorial: ▸ keras train and evalulate
- Source ▸ code

Outline

Save and Restore (without `tf.keras`)

Carnegie Mellon

`tf_save_restore`

```
1  model1 = Model1()
2  model2 = Model2()
3  opt = tf.train.AdamOptimizer()
4
5  # define a checkpoint object wrapping all objects in K-V type
6  checkpoint = tf.train.Checkpoint(model1=model1, model2=model2, opt=opt)
7  checkpoint_prefix = './ckpt' # directory to save
8
9  # save
10 checkpoint.save(checkpoint_prefix)  # ckpt-1.index, ckpt-1.data-00000-of-00001, ...
11
12 # normally restore the latest one
13 checkpoint.restore(tf.train.latest_checkpoint(checkpoint_path))
```

- Using `tf.train.Checkpoint` to wrap any object you have
- Can use validation score to save the best latest checkpoint

Manual Save/Restore with `tf.keras`

Carnegie Mellon

keras_manual_save_weights_only

```
1  model = create_model() # including compile()
2
3  # train
4  model.fit(x, y, ...)
5
6  # save
7  model.save_weights('./checkpoints/my_checkpoint')
8
9  # load later
10 # sometimes, create a new exact same architecture then load weights
11 model.load_weights('./checkpoints/my_checkpoint')
```

- Need to have an exact same architecture to load weights
- Save some space

Manual Save/Restore with `tf.keras` (2)

Carnegie Mellon

`keras_manual_save_whole`

```
1  model = create_model() # including compile()
2
3  # train
4  model.fit(x, y, ...)
5
6  # save
7  model.save('./checkpoints/my_checkpoint') # whole model
8  # or
9  model.save('./checkpoints/my_checkpoint.h5') # force to HDF5 format
10
11 # load
12 new_model = tf.keras.models.load_model('my_model.h5')
```

- Whole model means: architecture, weights and optimizer
- Restore is simpler: no need to have a scaffold to load weights into

Save with callback: ModelCheckpoint

Carnegie Mellon

model_checkpoint

```
callbacks = [
    tf.keras.callbacks.ModelCheckpoint(
        filepath='mymodel_{epoch}',
        save_best_only=True,
        monitor='val_loss', # only if this improves
        save_weights_only=True,
        save_freq='epoch',
        verbose=1)
]
model.fit(x_train, y_train,
          batch_size=64,
          epochs=30,
          #validation_data=(x_val, y_val)),
          #validation_feq=1,
          validation_split=0.3,
          callbacks=callbacks)
```

- Usual usage: save after one or a few epochs, or save best models

Save with callback: ModelCheckpoint

Carnegie Mellon

model_checkpoint

```
callbacks = [
    tf.keras.callbacks.ModelCheckpoint(
        filepath='mymodel_{epoch}',
        save_best_only=True,
        monitor='val_loss', # only if this improves
        save_weights_only=True,
        save_freq='epoch',
        verbose=1)
]
model.fit(x_train, y_train,
          batch_size=64,
          epochs=30,
          #validation_data=(x_val, y_val)),
          #validation_feq=1,
          validation_split=0.3,
          callbacks=callbacks)
```

- Usual usage: save after one or a few epochs, or save best models
- Same options: save only weights (`model.save_weights`) or the whole model (`model.save`)

Restore from ModelCheckpoint

Carnegie Mellon

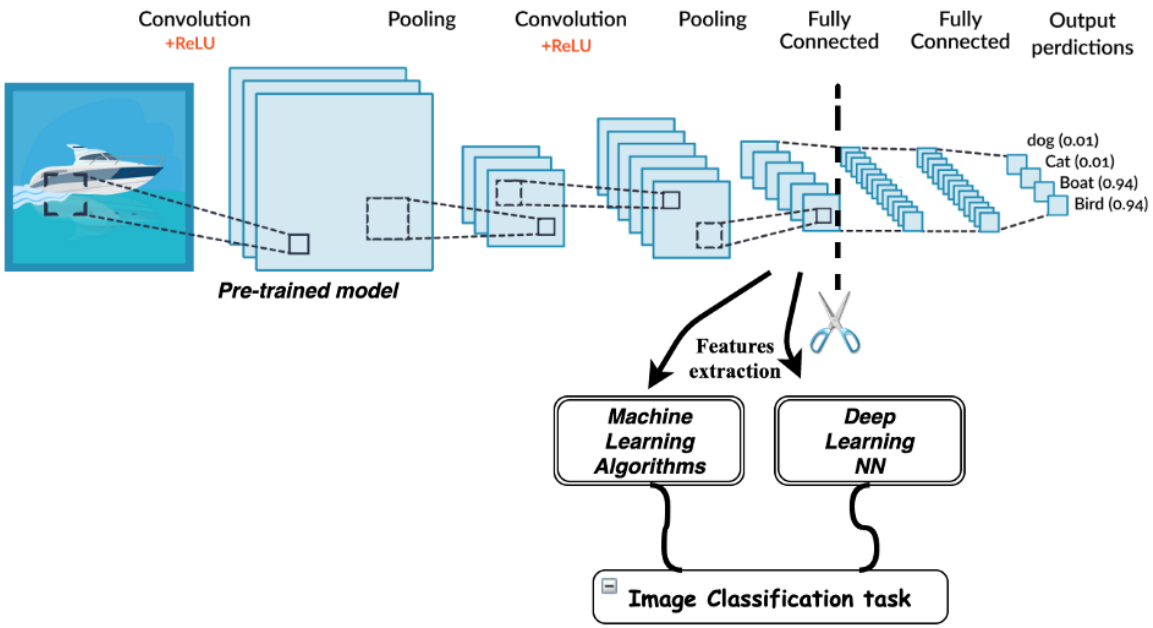
tf_keras_restore

```
# create an exact same model
model = Model() # must be the same structure

# restore
model.load_weights("mymodel.best.hdf5")

# compile and eval
model.compile(loss=..., optimizer=..., metrics=[...])
model.evaluate(x_test, y_test, verbose=1)
```

- Same use cases: restore from disk to evaluate a model or fine tune a pretrained model
- Same restore as manual operations above

Outline

Transfer Learning



- General use case: fine tuning with new yet similar data
- General techniques: re-use (ideally most) part of known knowledge into new data
- Implementation: re-use (ideally most) architecture and weights

Image credit: mc.ai

Load source model

Carnegie Mellon

keras_load_source

```
# create an exact same model and restore weights
source_model = Model() # must be the same structure
source_model.load_weights("mymodel.best.hdf5")
source_model.compile(loss=..., optimizer=..., metrics=[...])

# or load the whole model as a whole
source_model = tf.keras.models.load_model('my_model.h5')

# customize
... # soon later
```

- First is to load the source model with weights
- Techniques are the same as above for customized models

Load predefined source model[9]

Carnegie Mellon

keras_load_predefined_source

```
1  # Create the base model from the pre-trained model MobileNet V2
2  source_model = tf.keras.applications.ResNet50(input_shape=None,
3                                                 include_top=False,
4                                                 weights='imagenet')
5
6  # customize
7  ... # soon later
```

- `tf.keras.applications` has a list of predefined networks
- Weights are automatically downloaded when we load
- `input_shape` is inferred if None, or explicitly assigned
- Top FC layer can be optionally excluded

[9] https://github.com/keras-team/keras-applications/tree/master/keras_applications

Add more layers to source model

Carnegie Mellon

keras_add_layers

```
# add layers
global_average_layer = tf.keras.layers.GlobalAveragePooling2D()
prediction_layer = keras.layers.Dense(1)

# define new target model
target_model = tf.keras.Sequential([
      source_model,
      global_average_layer,
      prediction_layer
])

# compile
base_learning_rate = 0.0001
target_model.compile(optimizer=tf.keras.optimizers.RMSprop(lr=base_learning_rate),
                     loss=tf.keras.losses.BinaryCrossentropy(),
                     metrics=['accuracy'])
```

- Define new layer(s) and define a new model
- `keras.Sequential` accepts a model as a part

Add layers to arbitrary positions

Carnegie Mellon

keras_add_layers_random

```
# define new target model
target_model = tf.keras.Sequential()

for layer in source_model.layers[:-10]: # exclude last 10 layers from copying
    target_model.add(layer)

# add your new layers here
target_model.add(global_average_layer)
target_model.add(prediction_layer)

# compile
...
```

- Define a new empty model and copy layers
- Weights should have been loaded before copy

Add layers to arbitrary positions (2)

Carnegie Mellon

keras_add_layers_random_two

```
# define new target model
target_model = tf.keras.Model()

taken = model.layers[-10].output

# add your new layers here
global_average_layer = tf.keras.layers.GlobalAveragePooling2D()
prediction_layer = keras.layers.Dense(1)

# make them attached sequentially
out = global_average_layer(taken)
out = prediction_layer(out)

# define new model with tf.keras.Model
target_model = tf.keras.Model(tf.keras.layers.Input(shape=(224,224,3), out)

# compile
...
```

- Take output of the point we want
- And attach new layers sequentially

Training: Freeze pretrained weights first

Carnegie Mellon

freeze_source_weights

```
# freeze source weights which is part of target model
# should do before adding new layers
source_model.trainable = False

# define target_model
target_model = tf.keras.Sequential([
  source_model,
  global_average_layer,
  prediction_layer
])

# compile
target_model.compile(...)

# train new layers only
history = target_model.fit(x_train, y_train, ...) # we use history later

# eval
target_model.evaluate(x_test, y_test, ...)
```

- First, train new layers only

Training: Fine tune pretrained weights

Carnegie Mellon

fine_tune

```
# un-freeze source weights which is part of target model
source_model.trainable = True

# Freeze all the layers before the ‘fine_tune_at‘ layer
for layer in base_model.layers[:100]: # fine tune from 100th layer onwards
  layer.trainable =  False

# NOTE: compile again
target_model.compile(loss=tf.keras.losses.BinaryCrossentropy(),
                     optimizer = tf.keras.optimizers.RMSprop(lr=base_learning_rate/10),
                     metrics=['accuracy'])

# continue training
total_epochs = initial_epochs + fine_tune_epochs
history_fine = target_model.fit(train_batches,
                                epochs=total_epochs,
                                initial_epoch =  history.epoch[-1],
                                validation_data=validation_batches)
```

- Then, fine tune pretrained weights
- For deep CNN, should fine tunes more specialized layers only

More on Save/Restore

Carnegie Mellon

- Tutorial: ▸ tf.train.Checkpoint
- Tutorial: ▸ tf.keras-Save and Serialize
- Tutorial: ▸ tf.keras-Based Save and Restore
- Tutorial: ▸ SavedModel format
- Tutorial: ▸ Load pretrained model and add new layers