Intro

In Classical ML, the paradigm was

- construct, by hand, transformations of the input to alternatate representations
    - feature engineering: create representations corresponding to a "concept" useful for classificatin
    - we called this a pipeline
- after multiple transformations, the representation was good enough that a classifer could separate classes

In Deep Learning, the paradigm is very similar

- a sequence of transformations
    - each transformation is called a "layer"
        - "Deep Learning" is many layers of transformation
    - each layer successively constructs a new representation

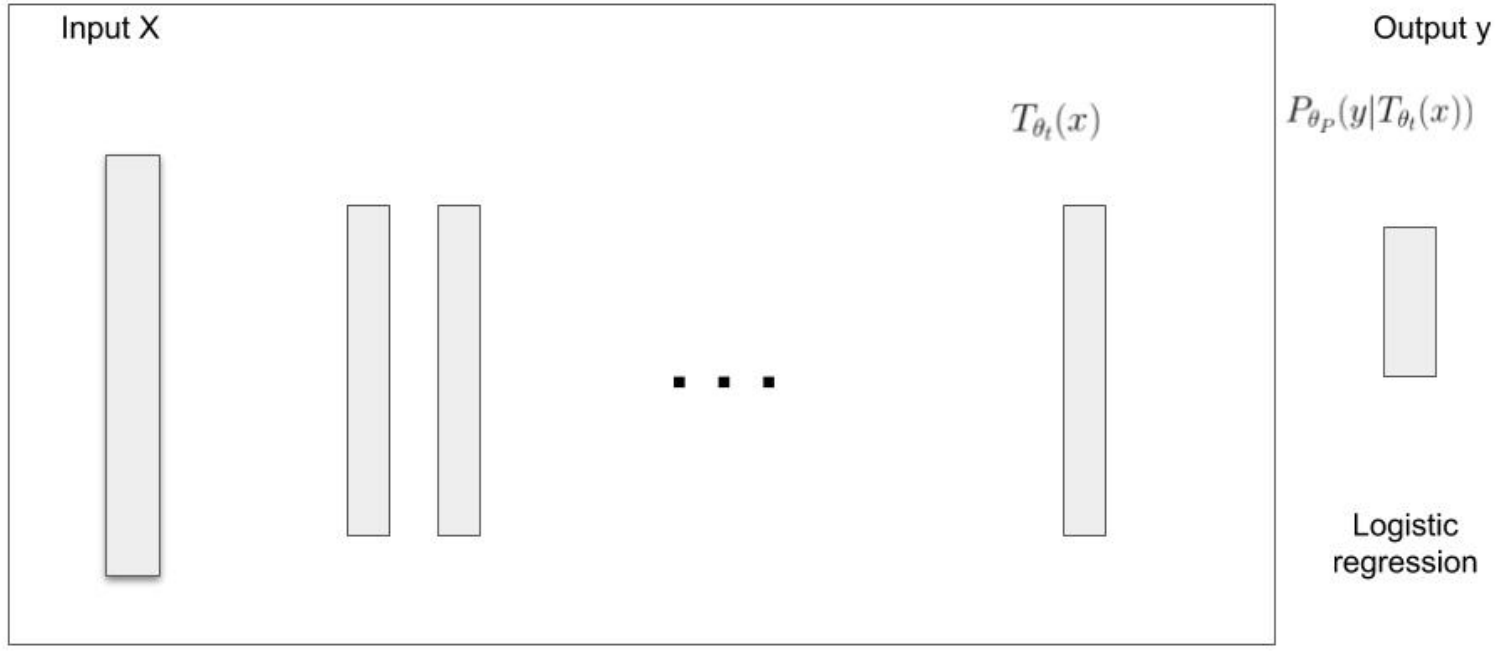
The key difference from Classical ML:

- the transformations are "discovered" rather than hand engineered

Deep Learning: the cartoon

Here is a "cartoon" diagram of Deep Learning (as applied to Classification)

Deep Learning: Classification



- the large vertical box at the left in the input (e.g., example: training or test)
- the smaller vertical boxes are "layers"
- the final layer (drawn outside the big box
    - is a layer that implements a Classifier
        - e.g., Logistic Regression

The input will be refered to as layer 0.

Layer $(l+1)$:

- takes as input: the output of layer l
- produces an output that is the transformed version of its input
    - the sizes of the inputs and outputs may be *very different*
    - determined by the internal workings of the layer

You can think of the output of each layer (inlcuding layer 0) as being alternate representations of our example's features.

So the NN is successively transforming the example's input features into different *representations*.

You can think of the represenation of layer $(l+1)$ as being more complex or "higher level" than that of layer l.

The goal of more complex representations is to eventually enable the final layer (e.g., Classifier) to be able to separate example classes.

Perhaps the lower level representations are not sufficient to enable the Classifier to succeed.

Our old friend, the dot product, will play a starring role.

- it is a fundamental part of many layers that we will study

Let's start with our the inputs to the NN: an example with our original features

Recall that the dot product can be thought of implementing a kind of "pattern matching:

- looking for a subset of features in an example

So one way to think of the transformations implemented by the **first layer** is that it

- recognizes patterns in the layer's input
- construct new "higher level" features

These higher level features may (or may not) be sufficient

- for a Classical Classifier to be able to correctly classify training examples
- for a Classical Regression to be able to correctly predict continous values

This also explains why we may need more than one layer

- layer $(l+1)$ takes as its input: the "high level feature" contructed by layer l
- with the benefit of a higher (compared to the training example) level features
    - we might match more complex patterns
    - and create even higher level features as output of layer $(l+1)$

Notation 1

Layer l:

- output of layer l : $\mathbf{y}_{(l)}$
  - subscripts in parenthesis refer to layer numbers
- input of layer l : $\mathbf{y}_{(l-1)}$
- layer 0 (*the input layer*) is where our examples are input
  - $\mathbf{y}_{(0)} = \mathbf{x}$

Let L denote the number of layers

- Final output of the NN:
  - $\mathbf{y} = \mathbf{y}_{(L)}$

How does the NN ”learn” the transformations ?

The answer is not too different than what we learned in Classical Machine Learning

- each layer has a set of parameters, e.g, $\Theta_{(l)}$ for layer l
- so the entire NN (the "model") has a collection of paramters Θ
- we define a per-example cost function
    - for our Classification or Regression problem
- we "solve" for the Θ that minimizes our Average (across examples) Cost

But how do we find the Cost-minimizing Θ ?

- In Classical ML: we sought closed-form solutions: Linear/Logistic Regression
- In Deep Learning: we will *search* for the best Θ
    - using search: we are able to use more complex Cost functions

Note

In DL, Θ is often refered to as the NN's *weights* and will thus often be denoted by $\mathbf{W}$ rather than Θ.

Training: finding the best $\mathbf{W}$

We begin by reviewing

- Cost functions
- Gradient Descent

Review

Cost/Loss, Utility/Optimization: review

- The prediction $\hat{\mathbf{y}}^{(i)}$ for example $\mathbf{x}^{(\mathbf{i})}$ is perfect if it matches the true label $\mathbf{y}^{(\mathbf{i})}$

$$\hat{\mathbf{y}}^{(\mathbf{i})} = \mathbf{y}^{(\mathbf{i})}$$

- The distance between $\hat{\mathbf{y}}^{(i)}$, $\mathbf{y}^{(\mathbf{i})}$ is called the *Loss* (or *Cost*) for example i:

$$\mathcal{L}_{\Theta}^{(\mathbf{i})} = L(\hat{\mathbf{y}}^{(\mathbf{i})}, \mathbf{y})$$

where $L(a, b)$ is a function that is 0 when $a = b$ and increasing as a increasingly differs from b.

Two versions L that we've seen are Mean Squared Error (for Regression) and Cross Entropy Loss (for classification).

The Loss for the entire training set is simply the average (across examples) of the Loss for the example

$$\mathcal{L}_{\Theta} = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}_{\Theta}^{(\mathbf{i})}$$

Whereas Loss describes how "bad" our prediction is, we sometimes refer to the converse -- how "good" the prediction is.

We call the "goodness" of the prediction the *Utility* U_{Θ}.

So we could state the optimization objective either as "minimize Cost" or "maximize Utility".

By convention, the DL optimization problem is usually framed as one of minimization (of cost or loss) rather than maximization of utility.

Gradient Descent review

- $\mathcal{L}_\Theta$ is a function of parameters Θ
- The (negative of) the derivative of $\mathcal{L}_\theta$ points in the direction of Θ that reduces
- One *step* of Gradient Descent updates Θ in the direction that reduces $\mathcal{L}_\Theta$

$$\Theta_{(t+1)} = \Theta_{(\mathbf{t})} - \alpha * \frac{\partial \mathcal{L}_\Theta}{\partial \Theta}$$

  where

  - subscript in parenthesis refers to the step number of Gradient Descent
    - update equation is for step $(t+1)$
  - α is the *learning rate*

Minibatch Gradient Descent review

$\mathcal{L}_\Theta$ is the sum of m per-example losses

$$\mathcal{L}_\Theta = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}_\Theta^{(\mathbf{i})}$$

- expensive to compute, for large m
- can *approximate* $\mathcal{L}_\Theta$ by evaluating it
    - across a *random subset (of size $m' \leq m$)* of examples: $I = \{i_1, \ldots, i_{m'}\}$

$$\mathcal{L}_\Theta \approx \frac{1}{|I|} \sum_{i \in I} \mathcal{L}_\Theta^{(\mathbf{i})}$$

Whereas Gradient Descent computes an exact $\mathcal{L}_\Theta$ to perform a single update of Θ:

Minibatch Gradient Descent

- takes $b = m/m'$ smaller steps, each updating Θ
- each small step using an approxmation of $\mathcal{L}_\Theta$ based on $m' \leq m$ examples

It does this by

- choosing batch size m'
- partitioning the set of example indices $\{i | 1 \leq i \leq m\}$
    - into b *batches* of size m'
    - batch $i' : b_{(i')}$ is one partition consisting of m' example indices
    - Each small step uses a single batch to approximate $\mathcal{L}_\Theta$ and update Θ

The collection of b small steps (comprising all examples) is called an *epoch*

So one epoch of Minibatch Gradient Descent performs b updates

The Training loop

The way we find the cost-minimizing Θ is by using Gradient Descent (or Minibatch Gradient Descent) on the Cost function $\mathcal{L}_\Theta$.

This is implemented in a loop called the *training loop*

During each iteration (epoch) t of the loop

- We compute $\hat{\mathbf{y}}^{(\mathbf{i})}$ for all m examples
    - Present $\mathbf{x}^{(\mathbf{i})}$ to the input layer of the NN
    - Evaluate $\hat{\mathbf{y}}^{(\mathbf{i})}$ using the current value for $\Theta : \Theta_{(t)}$
        - compute $\mathbf{y}_l$ for each layer $0 \leq l \leq L$
        - $\hat{\mathbf{y}}^{(\mathbf{i})} = \mathbf{y}_L$
- Each example i has it's per-example loss $\mathcal{L}_{\Theta}^{(\mathbf{i})}$

- The per-example losses contribute to a gradient (exact or approximate)
- The loss per batch causes a single update to Θ

So the Training Loop implements a Gradient Descent or one of its variants.

In "the old days" the **user** wrote the Training Loop.

Using a higher level API (like Keras) the loop is implemented for you in a single call.

This is called *training* or *fitting* the model.

We will examine both the high level API and the low level implementation.

initialize(Theta) # Training loop to implement mini-batch SGD for epoch in range(n_epochs):` for X_batch, y_batch in next_batch(X_train, y_train, batch_size, shuffle=True): # Forward pass z = NN(X_batch) # Loss calculation loss = loss_fn(z, y_batch) # Backward pass grads = gradient(loss, X) # Update Theta = Theta - grads * learning_rate

Layers

We will learn about several primitive types of layers:

- Dense/Fully Connected
- Activation
- Convolutional
- Recurrent
- Lambda
    - occassionally you will have to write your own operations
        - TensorFlow native operations: multiplication, addition

By "drawing a box" around a sequence of layers

- you can create a more complex layer
- same API, but hidden steps in the middle

Think of a Layer as a sequence of one or more

- Primitive layers (sequence of length 1, see above)
- Non-primitive layers
    - can treat a sequence of length greater than 1 as a single layer (treat the middle as a black box_

The Dense (Fully Connected) Layer

The first layer type we will learn about is called the *Dense* or $Fully Connected (FC)* layer.

Let layer l in the Sequential Model by a Dense layer

- It's inputs are $\mathbf{y}_{(l-1)}$ and outputs $\mathbf{y}_{(l)}$
- It consists of $n_{(l)}$ *neurons* (nodes)
    - So $|\mathbf{y}_{(l)}| = n_{(l)}$
- Each neuron j performs a dot product of
    - $\mathbf{y}_{(l)}$ and a set of per neuron parameters $\mathbf{W}_{(l),j}$
        - where $\mathbf{W}_{(l),j}$ is a vector of length $|\mathbf{y}_{(l)}|$
    - adds a constant per neuron *bias* $\mathbf{b}_j$

$$\mathbf{y}_{(l),j} = \mathbf{y}_{(l-1)} \cdot \mathbf{W}_{(l),j} + \mathbf{b}_j$$

To summarize Dense Layer l

- Its inputs are a representation (vector of features) of length $|\mathbf{y}_{(l-1)}|$
- It outputs a represenation of length $|\mathbf{y}_{(l)}|$

- The number of parameters (size of $\mathbf{W}_{(l)}$) is

$$|\mathbf{W}_{(l)}| = |\mathbf{y}_{(l)}| * (|\mathbf{y}_{(l-1)}| + 1)$$

  - one output for each of the $n_{(l)} = |\mathbf{y}_{(l)}|$ neurons
  - each neuron j has
    - $|\mathbf{y}_{(l-1)}|$ parameters to use in the dot product with $\mathbf{y}_{(l-1)}$
    - one bias $\mathbf{b}_j$

The Dense Layer is also called *Fully Connected*

- each output is "connected" to each input, via a single neuron.

Note

ALWAYS count the number of parameters

- They add up quickly !
- Dense Layers are particularly greedy
    - product of input size and output size

Dense Layer as pattern matching

Given an input representation, how do we tell whether it matches a certain feature ?

- weights as pattern
- dot product computes score
    - score is high if input is similar to pattern

So the Dense Layer l can be thought of as creating $|\mathbf{y}_{(l)}|$ "more complex" features.

We *don't* tell the NN what these features should be

- they are *discovered* (learned) by solving the Cost Function minimization

The fact that the new representation is learned rather than specified often makes the process of Deep Learning (and the intermediate represenations) it produces *difficult to interpret*.

We will make some attempts at interpretability throughout this course.

Although it is easiest to think of representations as one dimensional, that will rarely be the case

- images are a representation that is 2 or 3 dimensional (color channel)
- if a layer produces n features, its output volume has a "depth" of n
    - so the 3d image (2d plus one channel for each of 3 colors), after passed through a layer creating n features, now has depth n rather than 3

So best to think of "dot product" as

- element-wise multiplication of input representation with weights of same dimensionality
- reduce (sum) to scalar

Classical ML versus Deep Learning: MNIST digit recognition

Imagine we were trying to recognize MNIST digits in the Classical ML setting.

We might *hand-engineer* features using two levels of transormations before sending the new features to a Classifier

- layer 1: features
    - horizontal top
    - horizontal botton
    - diagonal, each direction
- layer 2: combo of layer 1 features
    - horizontal top and diagonal

This might seem like the way to do digit recognition were we to perform feature engineering by hand.

But our intuition of what features are useful may be off, or at least, sub-optimal

Deep Learning

- is an optimization that solves for patterns that prove most useful for classification.

Activation Layers

Observe that a Dense Layer is performing a linear transformation (because of the dot product).

From Linear Algera: the composition of linear transformations is itself a single linear transformation.

So there would be *no mathematical purpose* to a Sequential Model consisting of only Dense Layers.

Key aspect separating Classical ML from Deep LearningL

- DL can *also* implement *non-linear transformations*.

This turns out to be incredibly powerful.

In turn, a key in enabling this power is that we optimize the Cost Function by *search*

- closed form solutions for Non-Linear functions are hard !

Deep learning programming appends a "non-linear" *Activation Layer* at the end of every linear layer.

- there is a default Activation associated with each layer
- so, in practice, you specify the Activation layer as a parameter in constructing another layer
- know the default activation for your layer !

By doing so, the composition of layers is no longer linear.

Non-linearity allows us to compute arbitrary functions

- A NN is a Universal Function Approximator

The non-linearity is called an "activation" because it typically acts like a threshhold:

- if the linear "dot product score" is high, pass the score through to the ouptut
- otherwise reduce it to (near) 0

The activation introduces the ability to compute non-linear transformations as well as a mechanism to separate High scores ("has feature") form insignificant scores.

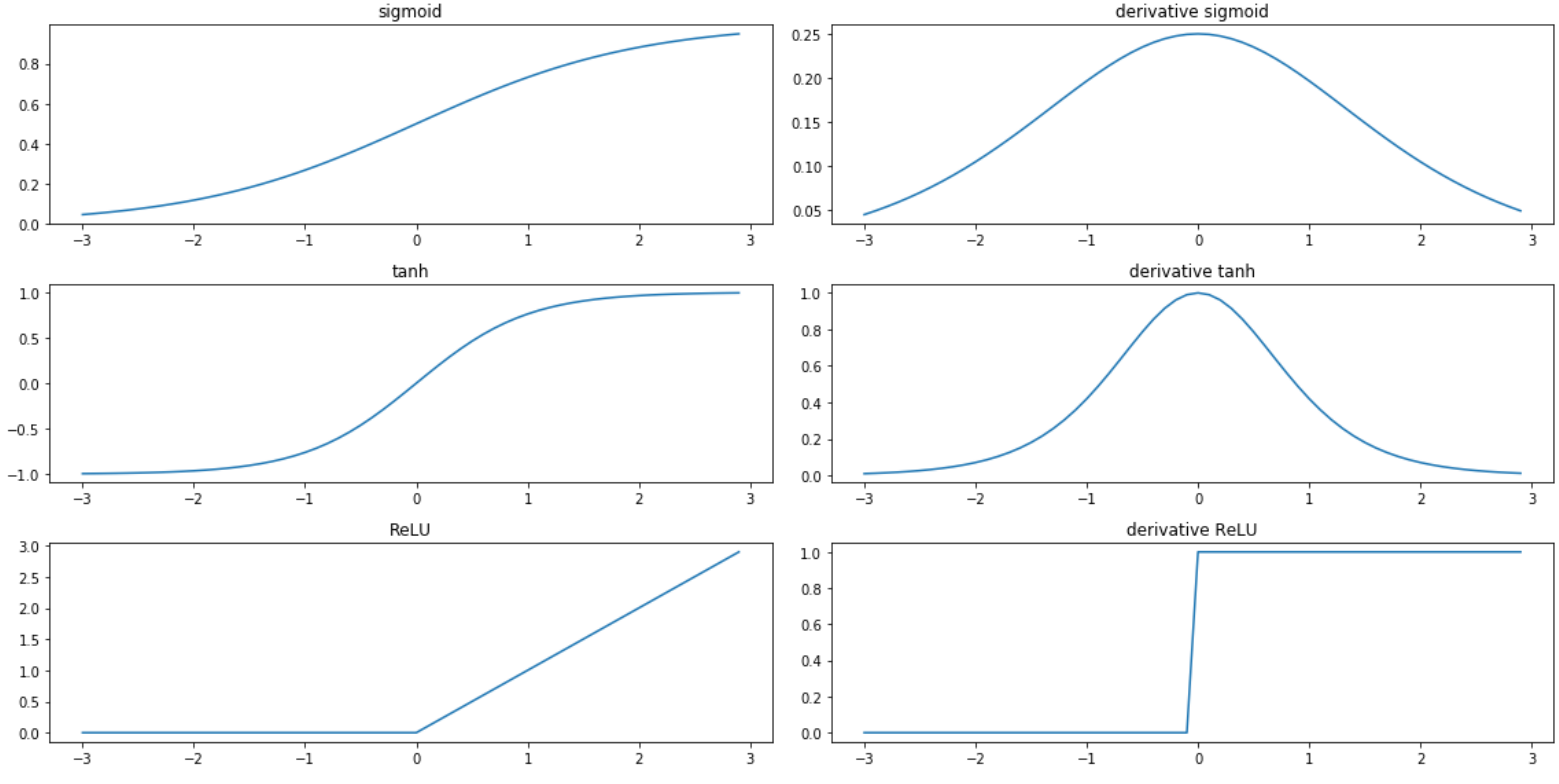
Some Activation Layers

There are several choices for the non-linear activation layer.

To get some intuition, let's show the plot of each, along with its derivative.

The derivative is **very important** as it will come into play during Gradient Descent.

In [4]:
```
fig, _ = nnh.plot_activations( np.arange(-3,3, 0.1) )
```



The first thing to note is the different output ranges.

The particular task might dictate the Activation function for the final layer

- the range of tanh is $[-1, +1]$ which may be appropriate for 0 centered outputs
- the range of sigmoid is $[0, 1]$, which may be appropriate for
    - binary classifiers, or neurons that act as "gates" (on/off switches)
    - outputs that need to be in this range, such as probabilities
- **No** activation might be the right choice for a Regression task (unbounded output range)

The other thing to notice are the derivatives:

- both the tanh and sigmoid have large regions, at either tail, of near zero derivatives
- the derivative of the sigmoid has a maximum value of about 0.25

Although it is hard to appreciate at the moment

- Managing derivatives is **one of the key** insights that enabled the explosive growth of DL !

We will explore this more in a subsequent lecture; for now:

- A zero derivative can hamper learning that uses Gradient Descent (tanh, sigmoid)
- The magnitude of the derivative modulates the "error signal" during back propagation
    - so smaller maximum values diminish the signal more than larger ones (sigmoid)

What does the bias do ?

The bias term (e.g., in the Dense layer) seems like an isolated annoyance.

Far from it !

The Activations map inputs in a fixed range to outputs.

The bias (of layer l) shifts the input to layer $(l+1)$.

So although, for example, the ReLU "hinge" is located at 0

- The bias of the previous layer can effectively shift this point.

Other activations

Softmax Layer

Leaky ReLU Layer

Loss "Layers"

In the L layer Sequential Model

- We sometimes see an additional layer ($L+1$) implementing the Cost/Loss function.

Although not truly a "layer" in the sense we have defined it

- It is not deriving a new representation
- It is a programming/mathematical convenience
    - makes it easier to implement and analyze Gradient Descent when the Loss is $\mathbf{y}_{(L+1)}$

Cost/Loss functions

We have framed Machine Learning (and DL in particular) as the task of minimizing a Cost/Loss function.

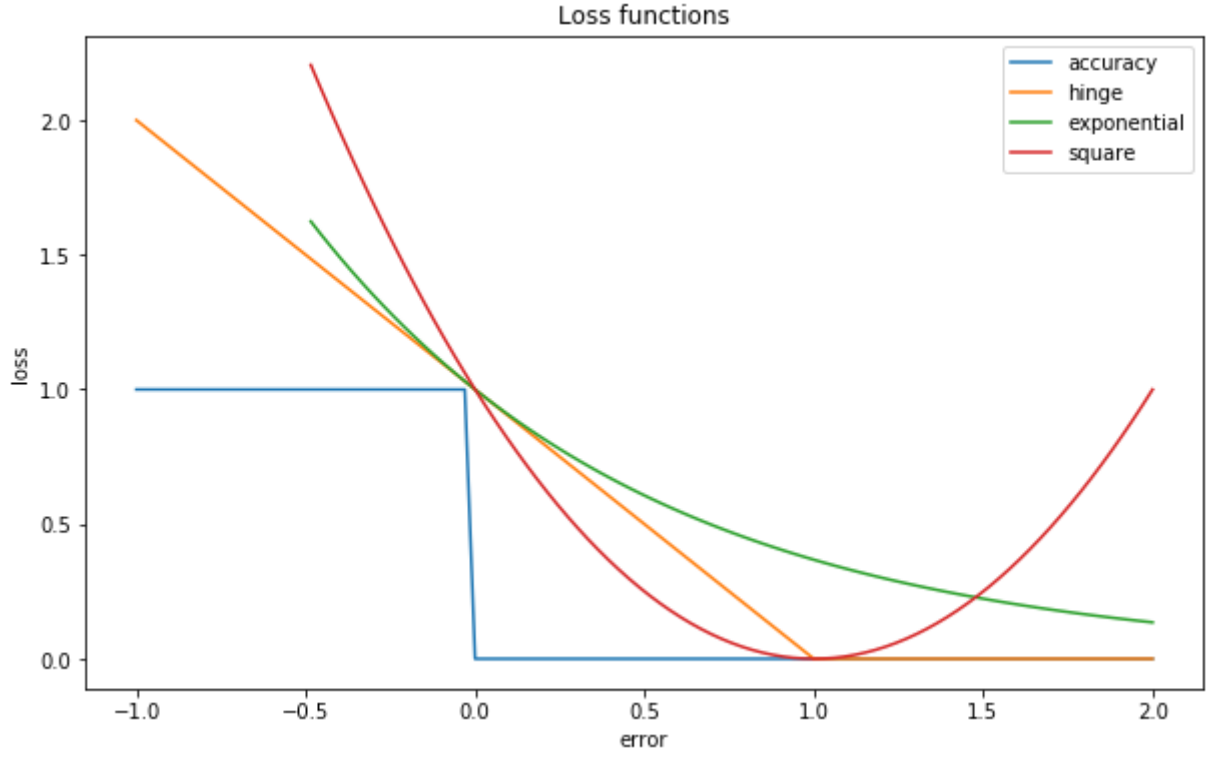
There are many such functions that we can consider.

We briefy discuss a few.

Let's begin by plotting some Cost functions

(Note: this statement is true regardless of whether the Loss is implemented as a layer.)

In [5]:
```
nnh.plot_loss_fns()
```



Overview

First, let's state some basic properties of a loss/function:

- it should be positive
- it should be close to 0 if the prediction is close to the correct value
- it should be **differentiable** (or at least sub-differentiable) if Gradient Descent is our optimizer

Accuracy

- The non-differentiability of Accuracy **rules it out** as a Cost function

Hinge Loss

We encountered this loss in the lecture on Support Vector Machines (SVM):

- It *is* sub-differntiable
- Is unusual in that there are elements in the domain where the Cost is *exactly* 0
- Gradient Descent will
    - **stop trying** to improve Θ, $\mathbf{W}$ once the Cost is 0
    - contrast this to Cross Entropy, where cost never reaches 0
        - Gradient Descent will keep trying to improve Θ, $\mathbf{W}$
        - Will try to increase score of already correct prediction
            - long after the probability threshold for Classification has been passed

Exponential loss

This is used in the AdaBoost (Adaptive Boosting) algorithm.

One can think of it as a smoothed version of either accuracy or hinge/margin loss.

- But never reaches a value of 0, so continous pressure to increase the score of an already correct prediction
- Asymmetric: there is greater pressure to correct an incorrect prediction than to increase the score of a correct prediction.

This measure can be used for either classification or regression.

When used for regression

- asymmetry implies a greater penalty for losses on one side of correct than the other.

Squared error

We have seen variants of this before (MSE) for regression.

In theory one can use this for classification

- But observe that there is a penalty (i.e., increased loss) for moving too far from the boundary.

So being "very confident" (higher score) is worse than being "just confident enough".

Complex loss functions: multiple objectives

Regularization objectives

Neural Style Transfer: fun with Cost functions

Neural style transfer (combine content and style)

- Objective function
  - Find output image, whose latent representation
    - in layer near input is close to “content” image’s latent representation at same layer
    - in deep layer(s) is close to “style” image’s latent representation at same layer

Content image | Style image | Output image

Scaling the inputs

Many times in this course we have pointed out that some models are *scale sensitive*.

Neural Networks are not *mathematically* sensitive but tend to be so *in practice*.

It is *highly recommended* to scale your data so their absolute values are around 1.0 or at least somewhat small.

Gradient Descent is the root of the problem:

- Two features on different scales can cause the optimizer to favor one over the other
- Activations can *saturate*
    - Output of dot product (Dense layer) is in the "flat* area of the activation
    - Zero derivative: no learning
- The Cost/Loss may be large in initial epochs when the target values are too different from the dot products
    - *Large* gradients: unstable learning
    - Weights are typically initialized to values less than 1.0, leading to small dot products

Remember: if you re-scale the inputs, you will need to invert the transformation when communicating the results

Universal function approximator

A Neural Network is a Universal Function Approximator.

This means that an NN that is sufficiently

- wide (large number of neurons per layer)
- and deep (many layers; deeper means the network can be narrower)

can approximate (to arbitrary degree) the function represented by the training set.

Recall that the training $\mathbf{X} = \{(\mathbf{x^{(i)}}, \mathbf{y^{(i)}}) | 1 \leq i \leq m\}$ is a set of input/target pairs.

This may look like a strange way to define a function

- but it is indeed a mapping from the domain of $\mathbf{x}$ (i.e., $\mathcal{R}^n$) to the domain of $\mathbf{y}$ (i.e., $\mathcal{R}$)
- subject to $\mathbf{y}^i = \mathbf{y}^j \text{ if } \mathbf{x}^i = \mathbf{x}^j$ (i.e., mapping is unique).

We give an intuitive proof for a one-dimensional function

- all vectors $\mathbf{x}$, $\mathbf{y}$, $\mathbf{W}$, $\mathbf{b}$ are length 1.

For simplicity, let's assume that the training set is presented in order of increasing value of $\mathbf{x}$, i.e.

$$\mathbf{x}^{(0)} < \mathbf{x}^{(1)} < \ldots \mathbf{x}^{(m)}$$

Consider a single neuron with a ReLU activation, computing

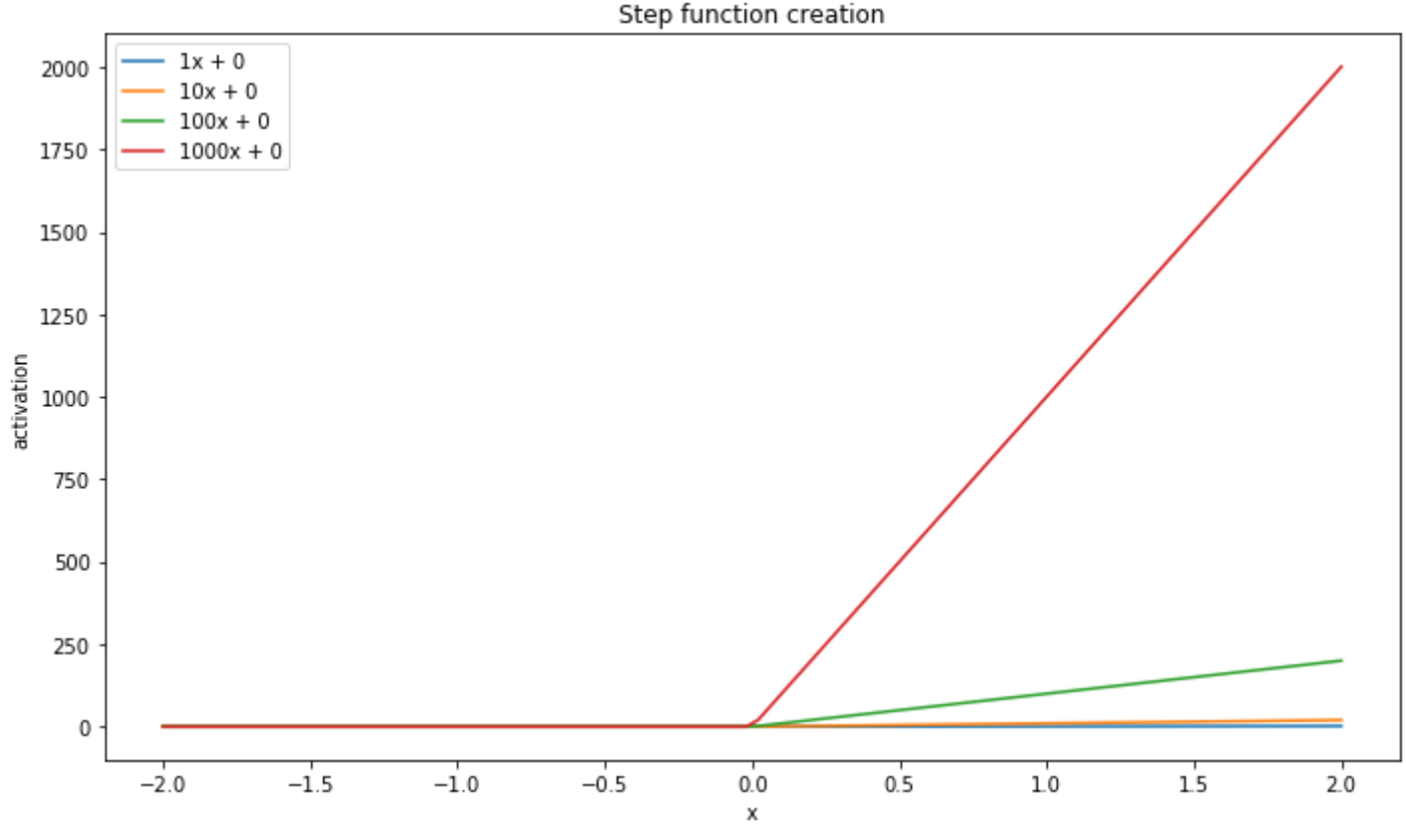
$$\max(0, \mathbf{W}\mathbf{x} + \mathbf{b})$$

Let's plot the output of this neuron, for varying $\mathbf{W}$, $\mathbf{b}$.

The slope of the neuron's activation is $\mathbf{W}$ and the intercept is $\mathbf{b}$.

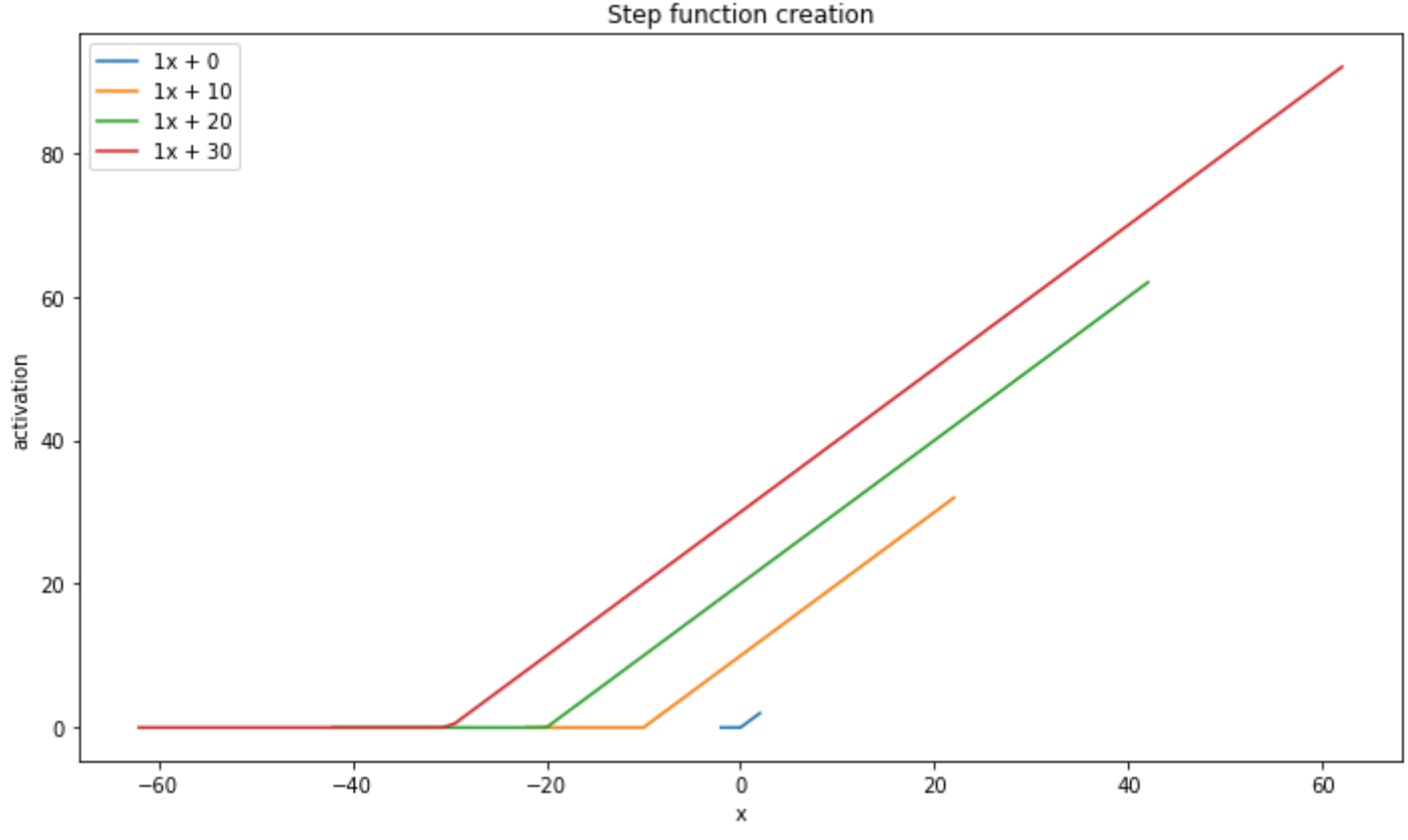
By making slope $\mathbf{W}$ extremely large, we can approach a vertical line.

In [6]:
```
_ = nnh.plot_steps( [ nnh.NN(1,0), nnh.NN(10,0), nnh.NN(100,0), nnh.NN(1000,0),
])
```



And by varying the intercept (bias) we can shift this vertical line to any point on the feature axis.

In [7]:
```
_ = nnh.plot_steps( [ nnh.NN(1,0), nnh.NN(1,10), nnh.NN(1,20), nnh.NN(1,30), ])
```



With a little effort, we can construct a neuron

- With near infinite slope
- Rising from the x-axis at any offset.

With a little more effort

- adding a neuron with an offsetting negative slope, at a very small distance from the x-intercept

we can construct an approximation of a step function

- unit height
- 0 output at inputs less than the x-intercept
- unit output for all inputs greater than the intercept).

(The sigmoid function is even more easily transformed into a step function).

We can construct neurons implementing step functions at any x-intercept.

- Construct m such neurons with intercepts $\{\mathbf{x^{(i)}} | 1 \leq i \leq m\}$
- Let us call the i^{th} such neuron "step neuron i".

If we connect the m step neurons to a "final" neuron with 0 bias, linear activation, and weights

$$\begin{aligned} \mathbf{W}_1 &= \mathbf{y}^{(1)} \\ \mathbf{W}_i &= \mathbf{y}^{(i)} - \mathbf{W}_{i-1} \end{aligned}$$

We claim that the output of this neuron approximates the training set.

To see this:

- Consider what happens when we input $\mathbf{x^{(i)}}$ to this network.
- The only step neurons that are active (non-zero) are those corresponding to inputs $1 \leq j \leq i$.
- The output of the final neuron is the sum of the outputs of the first i step neurons.
- By construction, this sum is equal to $\mathbf{y^{(i)}}$.

Thus, our neural network

- m step neurons
- Final "adder" neuron creates the mapping defined by the training set.

Financial analogy: if we have call options with completely flexible strikes and same expiry, we can mimic an arbitrary payoff in a similar manner.

Some "Why's"

Here are some interesting questions to ask about Deep Learning

Why was progress in DL so slow (decades) ?

We will answer this more deeply in a subequent lecture; for now:

- Seemingly minor details turned out to be incredibily important !
    - Initialization of weights Θ, $\mathbf{W}$ for Gradient Descent
    - Activations with large regions of zero gradient hampered learning
- Vanishing/Exploding Gradients
    - problems arise when the gradient is effectively 0
    - problems also occurs when they are effectively infinite

- Computational limits
    - It turns out to be quite important to make your NN big; bigger/faster machines help
    - Actually: bigger than it needs to be
        - many weights wind up near 0, which renders the neurons useless
        - The Lottery Ticket Hypothesis (https://arxiv.org/abs/1803.03635)
            - within a large network is a smaller, easily trained network
            - increasing network side increases the chance of large network containing a trainable subset
            - summary (https://towardsdatascience.com/how-the-lottery-ticket-hypothesis-is-challenging-everything-we-knew-about-training-neural-networks-e56da4b0da27)

Why do GPU's matter ?

What makes TPU's fined tuned for Deep Learning (https://cloud.google.com/blog/products/ai-machine-learning/what-makes-tpus-fine-tuned-for-deep-learning)

GPU (Graphics Processing Unit): specially designed hardware to perform repeated vector multiplications (a typical calculation in graphics processing).

- It is not general purpose (like a CPU) but does what it does extremely quickly, and using many more cores than a CPU (typically several thousand).
- As matrix multiplication is a fundamental operation of Deep Learning, GPU's have the ability to greatly speed up training (and inference).

Google has a further enhancement called a TPU (Tensor Processing Unit) to speed both training and inference.

- highly specialized to eliminate bottlenecks (e.g., memory access) in fundamental Deep Learning matrix multiplication.

Both GPU's and TPU's

- Incur an overhead (a "set up" step is needed before calculation).
- So speedup only for sufficiently large matrices, or long "calculation pipelines" (multiplying different examples by the same weights).

DL involves

- Multiplying large matrices (each example)
- By large matrices (weights, which are same for each example in batch)

- Both GPU's and TPU's offer the possibility of large speed ups.

- GPU's are **not** necessary

    - but they are a **lot** faster
    - life changing experience
        - 30x faster means your 10 minute run (that ended in a bug) now only takes 20 seconds
        - increases your ambition by faster iteration of experimental cycle

In [8]:
```
print("Done")
```

```
Done
```