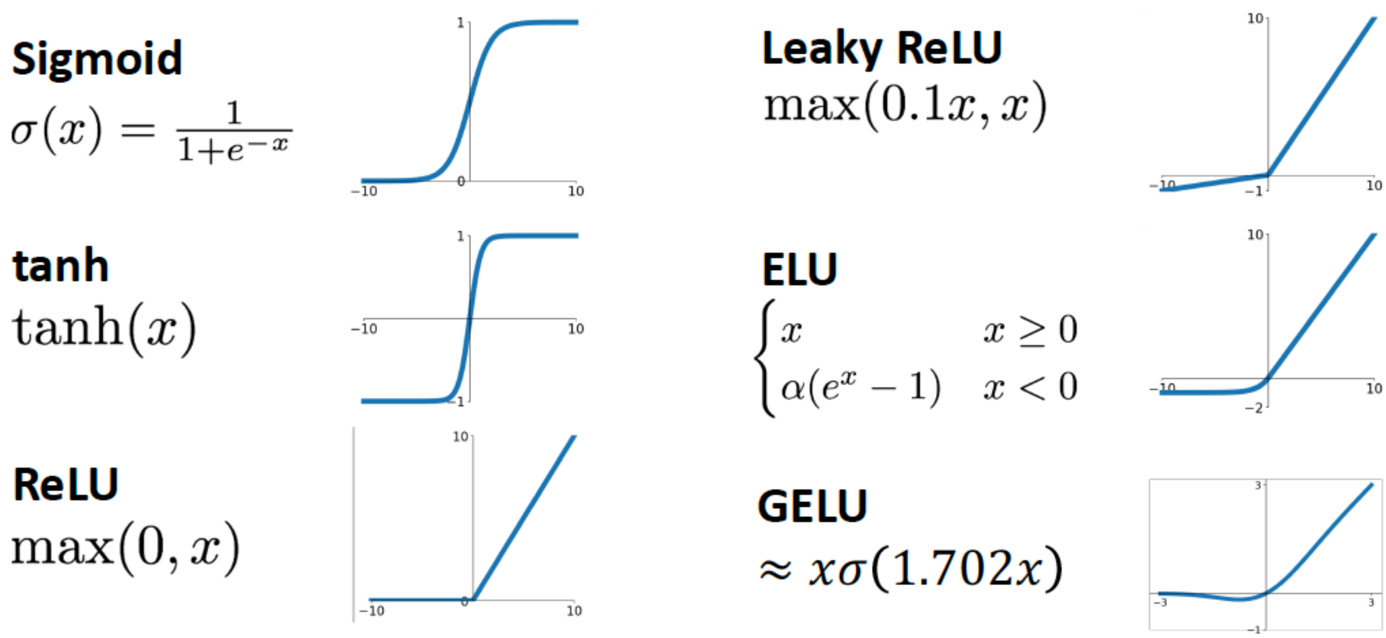
Lecture 9+10 - Training Neural Networks

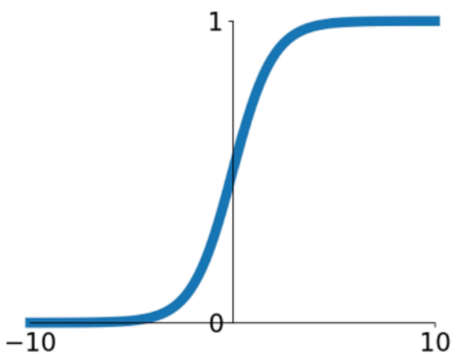
1. **One time setup**
   Activation functions, data preprocessing, weight initialization, regularization
2. **Training dynamics**
   Learning rate schedules; large-batch training; hyperparameter optimization
3. **After training**
   Model ensembles, transfer learning

1. Activation Functions



- **Sigmoid** **(Not use!)**

Activation Functions: Sigmoid



Sigmoid

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

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems: **Worst problem in practice**

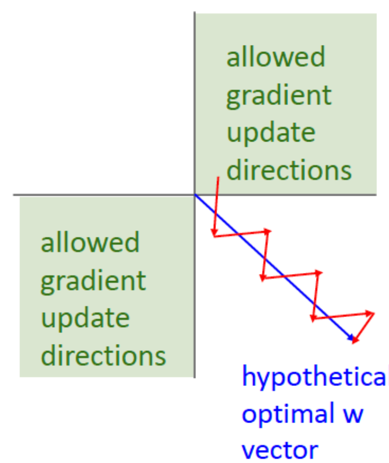
1. **Saturated neurons "kill" the gradients**
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

--> Sigmoid outputs are always positive

Consider what happens when nonlinearity is always positive

$$h_i^{(\ell)} = \sum_j w_{i,j}^{(\ell)} \sigma\left(h_j^{(\ell-1)}\right) + b_i^{(\ell)}$$

$h_i^{(\ell)}$ is the ith element of the hidden layer at layer ℓ (before activation)
$w^{(\ell)}, \mathrm{b}^{(\ell)}$ are the weights and bias of layer ℓ



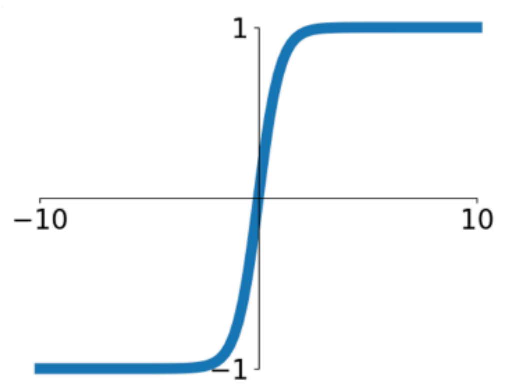
What can we say about the gradients on $w^{(\ell)}$?
Gradients on all $w_{i,j}^{(ell)}$ have the same sign as upstream gradient $\partial L / \partial h_i^{(\ell)}$

Not that bad in practice:
- Only true for a single example, minibatches help
- BatchNorm can also avoid this

--> all positive / negative --> want zero-mean data!

- **Tanh --> scaled and shifted Sigmoid (Not use!)**

Activation Functions: Tanh



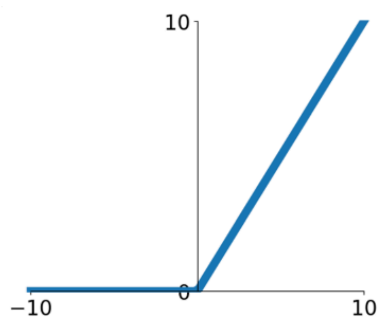
tanh(x)

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

- **ReLU**

Activation Functions: ReLU

$f(x) = \max(0,x)$

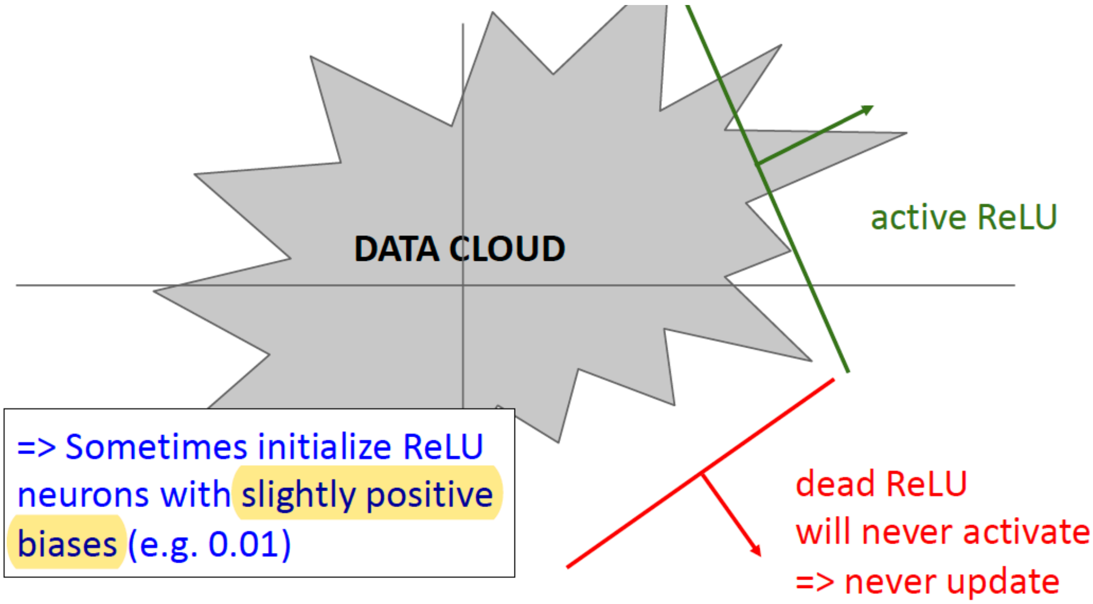


ReLU
(Rectified Linear Unit)

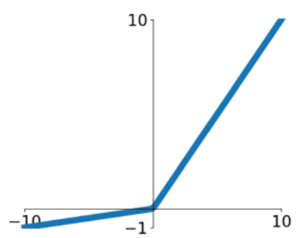
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

- Not zero-centered output
- An annoyance:

hint: what is the gradient when x < 0?
There will be no gradient update (0).



- **Leaky ReLU**

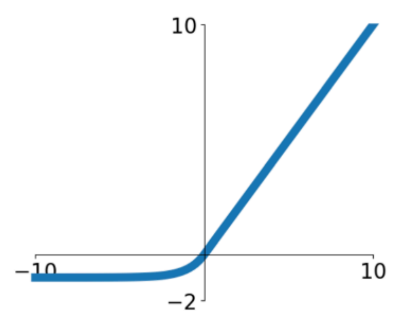


Leaky ReLU
$f(x) = \max(\alpha x, x)$
α is a hyperparameter, often $\alpha = 0.1$

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die".**

Parametric ReLU (PReLU)
$f(x) = \max(\alpha x, x)$
α is learned via backprop

- **Exponential Linear Unit (ELU)**



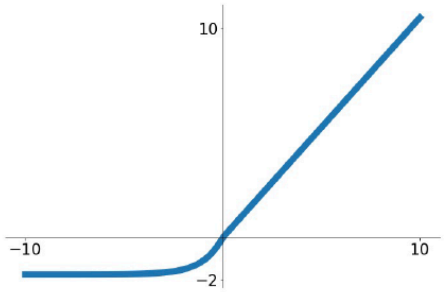
$$f(x) = \begin{cases} x & if\ x > 0 \\ \alpha(e^x - 1) & if\ x \le 0 \end{cases}$$

(Default alpha=1)

- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise
- Not get "dead ReLU" problem so much

Computation requires exp()

- **Scaled Exponential Linear Unit (SELU)**



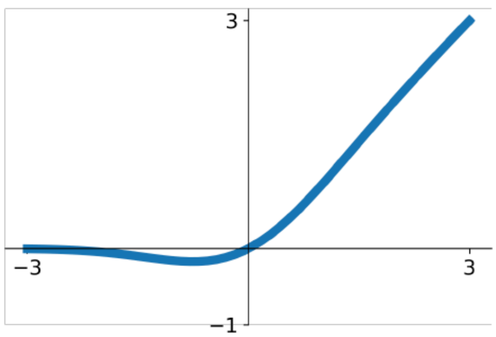
- Scaled version of ELU that works better for deep networks
- "Self-Normalizing" property; can train deep SELU networks without BatchNorm

$$selu(x) = \begin{cases} \lambda x & if\ x > 0 \\ \lambda\alpha(e^x - 1) & if\ x \leq 0 \end{cases}$$

$$\alpha = 1.6732632423543772848170429916717$$
$$\lambda = 1.0507009873554804934193349852946$$

Klambauer et al, Self-Normalizing Neural Networks, ICLR

- **Gaussian Error Linear Unit (GELU)**

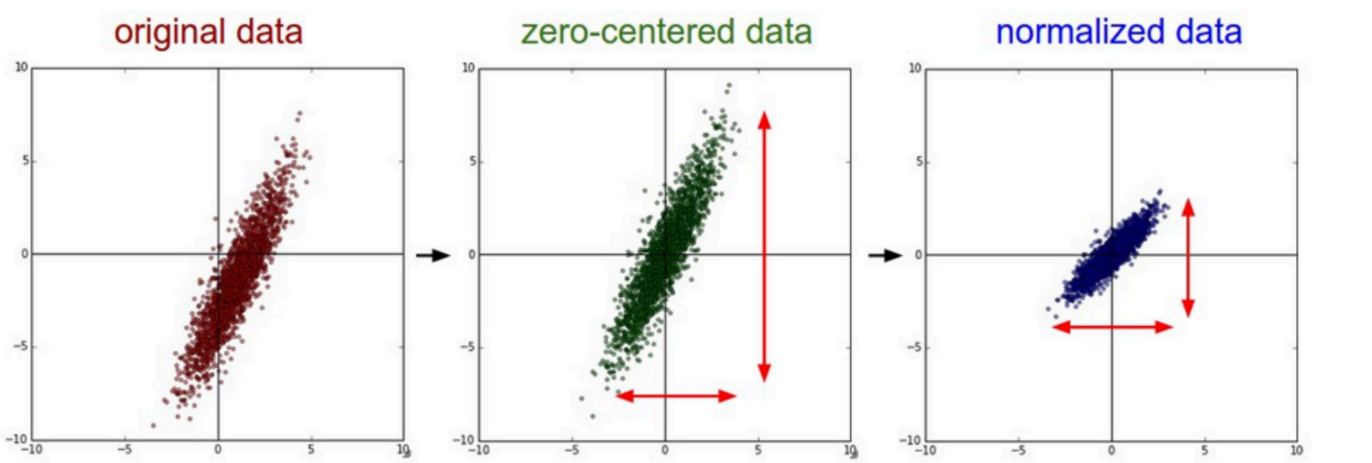


$$X \sim N(0,1)$$
$$gelu(x) = xP(X \leq x) = \frac{x}{2}\left(1 + \mathrm{erf}\left(x/\sqrt{2}\right)\right)$$
$$\approx x\sigma(1.702x)$$

- **Idea**: Multiply input by 0 or 1 at random; large values more likely to be multiplied by 1, small values more likely to be multiplied by 0 (data-dependent dropout)
- Take expectation over randomness
- Very common in Transformers (BERT, GPT, ViT)

Hendrycks and Gimpel, Gaussian Error Linear Units (GELUs), 2016

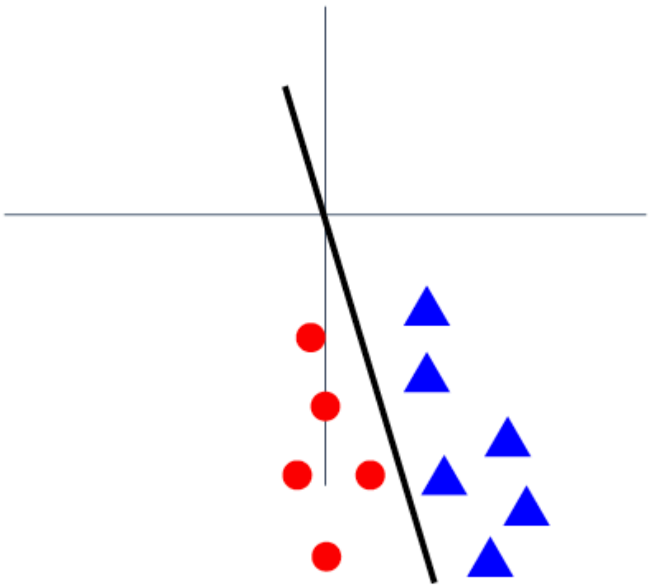
2. Data Preprocessing



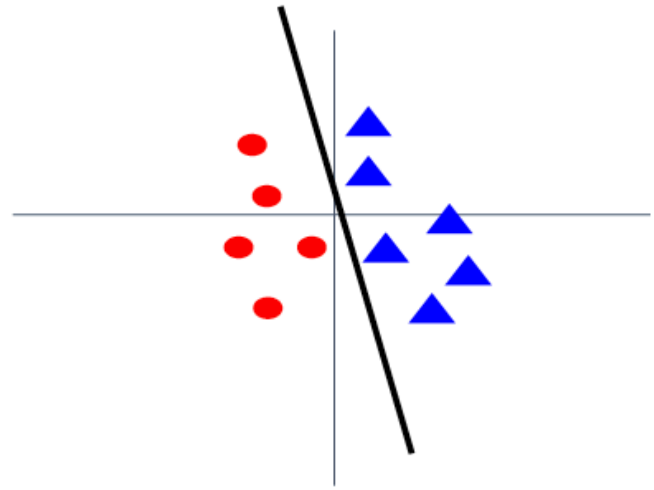

```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize

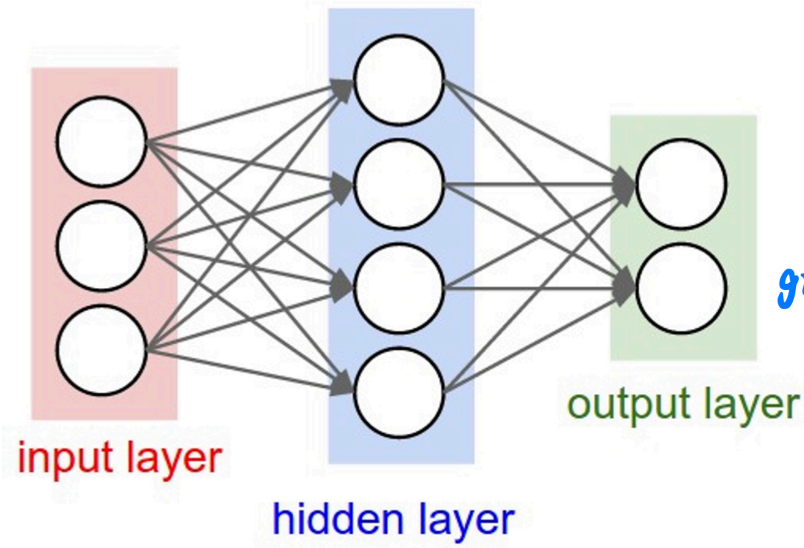


e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)
- Subtract per-channel mean and
  Divide by per-channel std (e.g. ResNet)
  (mean along each channel = 3 numbers)

Not common to do PCA or whitening

3. Weight Initialization



Q: What happens if we initialize all W=0, b=0?

hidden layers = 0

A: All outputs are 0, all gradients are the same!

gradient descent = 0

No "symmetry breaking"

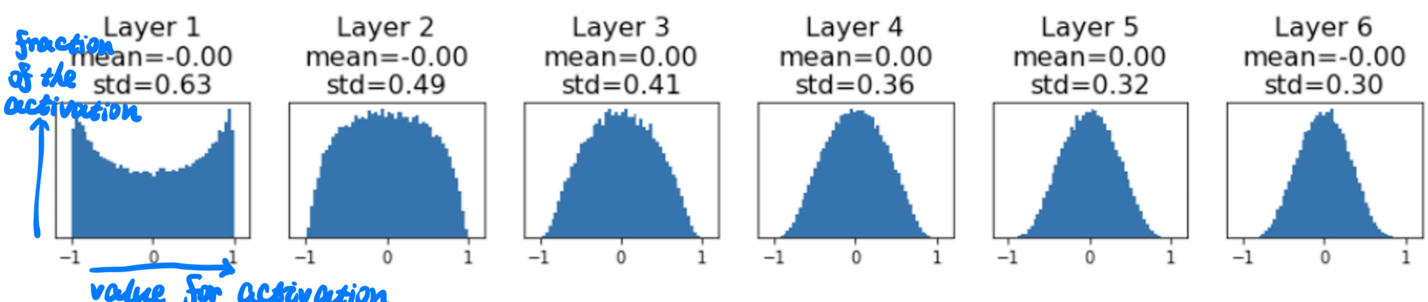
"Xavier" initialization:
std = 1/sqrt(Din)

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Just right": Activations are nicely scaled for all layers!

=> nor too small nor too large

For conv layers, Din is $kernel\_size^2$ * input_channels



Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010