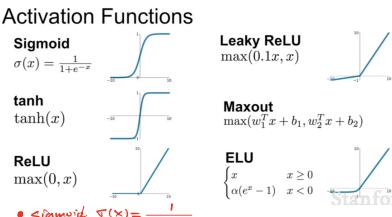


- Recall NN, CNN
  - ↳ activation map
  - ↳ objective: learn the weights by optimization
- mini-batch Stochastic Gradient Descent
  - Loop:
    1. sample a batch of data
    2. forward prop it thru the graph (compute) get loss
    3. backprop to calculate the gradients
    4. update parameters using gradients

- In this lecture:
- 1. one time setup:
  - activation functions, pre-processing, weight initialization, gradient checking
- 2. training dynamics:
  - batching, learning process, parameters update, hyperparameter optimization
- 3. evaluation model ensembles

## 1. Activation Functions



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

- squash numbers to range [0, 1]
- historically popular since they have nice interpretation as a "squashing fitting rate" of a neuron

→ problem:
 

- I. saturated neurons "kill" the gradients

What happens when  $x = -10^9$ ? What happens when  $x = 0^9$ ? What happens when  $x = 0$ ?  
2 sigmoid outputs are not zero-centered

Consider what happens when the input to a neuron ( $x$ ) is always positive:

$$f\left(\sum_i w_i x_i + b\right)$$

What can we say about the gradients on  $w_2$ ? Always all positive or all negative? (this is also why you want zero-mean data)

Consider what happens when the input to a neuron is always positive...

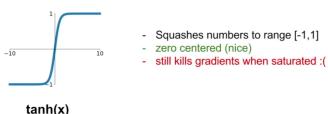
What can we say about the gradients on  $w_2$ ? Always all positive or all negative? (this is also why you want zero-mean data)

[the slow flow of sigmoid activation layer is not zero-mean: indicating next layer will create a problem for the gradients on W shown above]

3. exp() is a bit computation expensive

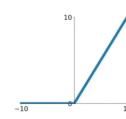
## • tanh

### Activation Functions



## • ReLU

### Activation Functions



#### ReLU (Rectified Linear Unit)



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

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

- Not zero-centered
- what is the gradient

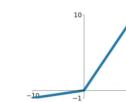
$$\begin{aligned} x = -10 &\Rightarrow 0 \\ x = 0 &\Rightarrow 0 \\ x = 10 &\Rightarrow 1 \end{aligned}$$

ReLU. The Rectified Linear Unit has become very popular in the last few years. It computes the function  $f(x) = \max(0, x)$ . In other words, the activation is simply thresholded at zero (see image above on the left). There are several pros and cons to using the ReLUs:

- (1) It was found to greatly accelerate (e.g. a factor of 6 in Krizhevsky et al.) the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form.
- (4) Compared to tanh/sigmoid networks that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero.
- (3) Unfortunately, ReLU units can be fragile during training and can "die". For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training since they can get knocked off the data manifold. For example, you may find that as much as 40% of your network can be "dead" (i.e. neurons that never activate across the entire training dataset) if the learning rate is set too high. With a proper setting of the learning rate this is less frequently an issue.

## • Leaky ReLU

### Activation Functions



[Mass et al., 2013]  
[He et al., 2015]

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

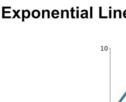
#### Parametric rectifier (PReLU)

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

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

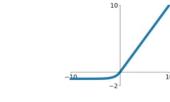
## • ELU

### Activation Functions



[Clevert et al., 2015]

#### Exponential Linear Units (ELU)



- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

- Computation requires exp()

## • Maxout

### Maxout "Neuron"

[Goodfellow et al., 2013]

- Does not have the basic form of dot product → nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

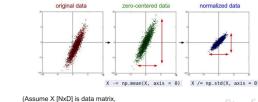
Problem: doubles the number of parameters/neuron :

## TLDL:

- △ use ReLU (learning rates matter!)
- △ try Leaky ReLU/Maxout/ELU
- △ try tanh yet don't expect much
- △ don't use sigmoid!!!

## • Step 1. Preprocessing the data

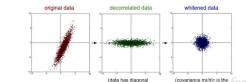
### Step 1: Preprocess the data



Stanfor

### Step 1: Preprocess the data

In practice, you may also see PCA and Whitening of the data



Stanfor

TLDR: In practice for Images, center only → take mean  
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

L, R, B → Not common to normalize variance, to do PCA or whitening

### • weight initialization

- Q: What will happen if  $W \in \mathbb{R}^{n \times d}$ ?
- A: neurons will learn the same thing

#### method 1: small random numbers

- e.g.  $w \sim 0.01 \cdot \text{np.random.randn}(n, d)$
- ok for small network
- will die out

#### method 2: larger random numbers.

- will saturate

#### method 3: Xavier Initialization

### Batch Normalization

#### Batch Normalization

you want gaussian activations? just make them so.

consider a batch of activations at some layer. To make each dimension unit gaussian, apply:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

this is a vanilla function...  
do backprop

Batch Normalization

you want unit gaussian activations? just make them so.

1. compute the empirical mean and variance independently for each dimension.
2. Normalize

Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

Problem: do we need to store and gaussian value for each dimension?

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

Stanford

[Joffe and Szegedy, 2015]

Batch Normalization

Normalize:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

Note: the network can learn:

$$\gamma^{(k)} = \sqrt{\text{Var}[x^{(k)}]} \quad \beta^{(k)} = \mathbb{E}[x^{(k)}]$$

to recover the identity mapping.

[Joffe and Szegedy, 2015]

Batch Normalization

Report: Values of  $x$  are a mini-batch:  $B = \{x_1, \dots, x_n\}$ .

Output:  $\{y_1, \dots, y_n\} = \text{BN}_{\gamma, \beta}(B)$

$\mu_B = \frac{1}{n} \sum_{i=1}^n x_i$

$\sigma_B^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \mu_B)^2$

$y_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2}} + \beta$

$y_i = \gamma \hat{x}_i + \beta$

minibatch mean

minibatch variance

scale and shift

improve gradient flow through the network

Allows higher learning rates on the network

Reduces the strong dependence on initial values

Act as a form of regularization

In a funny way, and slightly interestingly, it reduces the need for Dropout, maybe

[Joffe and Szegedy, 2015]

Balancing the training { train }

Hyperspectral optimization { weight }

cross-validation { dataset }

[Joffe and Szegedy, 2015]

## - Training II

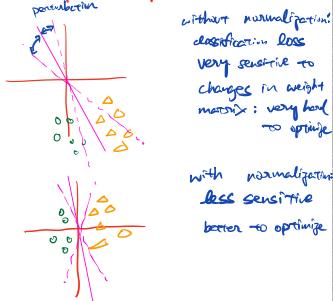
- recall:

① always/or mostly: use ReLU

② weight initialization:

- initialization too small:  
activations go to zero,  
gradients also zero,  
no learning.)
- initialization too big:  
activations saturate  
(for tanh)  
gradients zero, no learning
- initialization just right:  
(nice distribution of  
activations at all layers,  
learning proceeds nicely)

③ Data processing



## Batch Normalization

Input:  $x : N \times D$

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Learnable params:

$$\gamma, \beta : D$$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

Intermediates:

$$\hat{x} : N \times D$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Output:  $y : N \times D$

$$y_{i,j} = \gamma \hat{x}_{i,j} + \beta_j$$

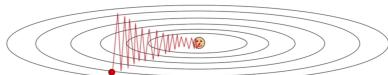
## △ Fancier optimization

### ① Optimization: Problems with SGD

What if loss changes quickly in one direction and slowly in another?

What does gradient descent do?

Very slow progress along shallow dimension, jitter along steep direction

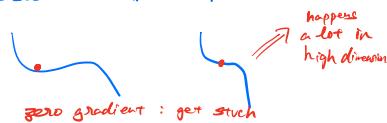


Loss function has high condition number: ratio of largest to smallest singular value of the Hessian matrix is large

Stanford

even more severe in high dimensions

### ② local minima // saddle point



③

### Optimization: Problems with SGD

Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^N \nabla_W L_i(x_i, y_i, W)$$

