- Recall NN, CNN

img → activation map

objective : learn the weights
by : optimization

- mini-batch Stochastic Gradient Descent

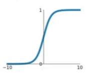
  Loop:
  1. sample a batch of data
  2. forward prop it thru the graph (network)
     get loss
  3. backprop to calculate the gradients
  4. update parameters using gradients

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

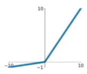
1. activation functions

Activation Functions

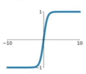
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



Leaky ReLU
$\max(0.1x, x)$

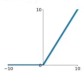


tanh
$\tanh(x)$

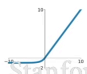


Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ReLU
$\max(0, x)$

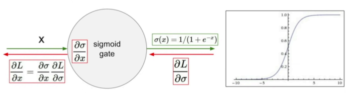


ELU
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$



- sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$
  - squashes numbers to range [0, 1]
  - historically popular since they hv nice intepretation as a saturating "fitting rate" of a neuron

→ problem:
  1. saturated neurons "kill" the gradients

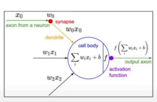


What happens when x = -10? $\frac{\partial \sigma}{\partial x} \approx 0$ (no gradient)
What happens when x = 0?
What happens when x = 10? $\frac{\partial \sigma}{\partial x} \approx 0$

  2. sigmoid outputs are not zero-centered

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

input = $w_0 x_0$, $w_1 x_1$, $w_2 x_2$, ⋮

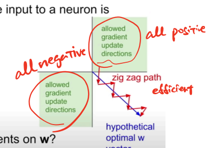


$$f\left(\sum_i w_i x_i + b\right)$$

What can we say about the gradients on **w**? is always all positive or all negative.

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

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



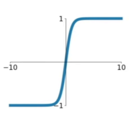
What can we say about the gradients on **w**?
Always all positive or all negative :(
(this is also why you want zero-mean data!)

(the data flow out of sigmoid activation layer is not zero mean : indicating @ next layer or later it will create a problem for the gradients on W shown above)

  3. exp() is a bit computation expensive

- tanh

Activation Functions

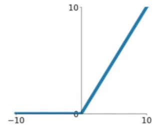


tanh(x)

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

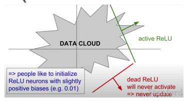
- ReLU

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
  x = -10 ⇒ 0
  x = 0 ≜ 0
  x = 10 ⇒ 1



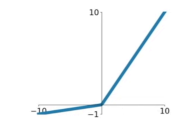
ReLU. The Rectified Linear Unit has become very popular in the last few years. It computes the function $f(x) = \max(0, x)$. In other words, the activation is simply thresholded at zero (see image above on the left). There are several pros and cons to using the ReLUs:

- (+) 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.
- (+) Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero.
- (-) 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]



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

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

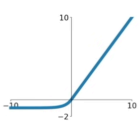
parametric rectifier (PReLU)
$f(x) = \max(\alpha x, x)$

- 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

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

- 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!

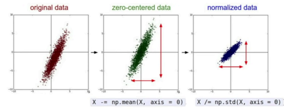
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Problem: doubles the number of parameters/neuron :(

TLDR:
△ 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

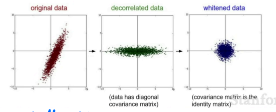
Step 1: Preprocess the data



(Assume X [NxD] is data matrix, each example in a row)

Step 1: Preprocess the data

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



mean of all training images

TLDR: In practice for Images: center only
take zero 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)
  R, G, B

→ Not common to normalize variance, to do PCA or whitening

- weight initialization

  - Q: What will happen if init W = 0
    A: neurons will learn the same thing

  △ method 1: small random numbers
    e.g. w = 0.01 * np.random.randn(D,H)
    ok for small network
    → will fade out

  △ method 2: larger random numbers.
    → will saturated

  △ method 3: Xavier initialization

- Batch Normalization

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want unit gaussian activations? just make them so."

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

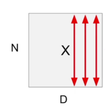
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...
able to do backprop

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want unit gaussian activations?
just make them so."



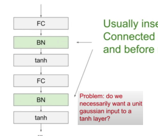
1. compute the empirical mean and variance independently for each dimension.

2. Normalize

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization

[Ioffe and Szegedy, 2015]



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

Problem: do we necessarily want a unit gaussian input to a tanh layer?

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

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

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

Note, the network can learn:

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

to recover the identity mapping.

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$ // mini-batch mean

$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$ // mini-batch variance

$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$ // normalize

$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$ // scale and shift

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

- Babysitting the training
- Hyperparameter optimization
  } train it urself (after lecture!)
- cross-validation

— Training II

— recall:

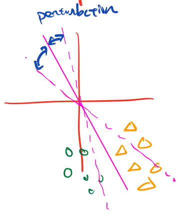
① always/or mostly: use ReLU

② weight initialization:

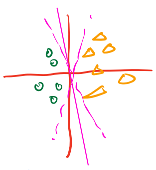
- initialization too small (activations go to zero, gradients also zero, no learning)
- initialization too big: (activations saturate (for tanh) gradients zero, no learning
- initialization just right (nice distribution of activations at all layers, learning proceeds nicely)

③ Data processing

perturbation



without normalization: classification loss very sensitive to changes in weight matrix: very hard to optimize



with normalization: less sensitive better to optimize

Batch Normalization

Input: $x : N \times D$

Learnable params: $\gamma, \beta : D$

Intermediates: $\mu, \sigma : D$, $\hat{x} : N \times D$

Output: $y : N \times D$

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

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

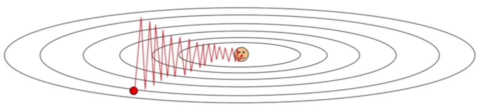
$$\hat{x}_{i,j} = \frac{x_{i,j} - \sigma_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

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

△ 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

even more severe in high dimensions

② local minima || saddle point

happens a lot in high dimension

zero gradient : get stuck

③

Optimization: Problems with SGD

Our gradients come from minibatches so they can be noisy!

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

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