Applied Machine Learning

Multilayer Perceptron

Siamak Ravanbakhsh

COMP 551 (winter 2020)

Learning objectives

multilayer percepron:

- model
  - different supervised learning tasks
  - activation functions
  - architecture of a neural network
- its expressive power
- regularization techniques

Adaptive bases

several methods can be classified as *learning these bases adaptively*

$$f(x) = \sum_d w_d \phi_d(x; v_d)$$

- decision trees
- generalized additive models
- boosting
- neural networks ←

Adaptive bases

several methods can be classified as *learning these bases adaptively*

$$f(x) = \sum_d w_d \phi_d(x; v_d)$$

- decision trees
- generalized additive models
- boosting
- neural networks
  - consider the adaptive bases in a general form (contrast to decision trees)

Adaptive bases

several methods can be classified as *learning these bases adaptively*

$$f(x) = \sum_d w_d \phi_d(x; v_d)$$

- decision trees
- generalized additive models
- boosting
- neural networks
  - consider the adaptive bases in a general form (contrast to decision trees)
  - use gradient descent to find good parameters (contrast to boosting)

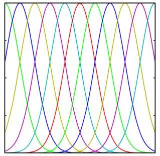
Adaptive bases

several methods can be classified as *learning these bases adaptively*

$$f(x) = \sum_d w_d \phi_d(x; v_d)$$

- decision trees
- generalized additive models
- boosting
- neural networks
  - consider the adaptive bases in a general form (contrast to decision trees)
  - use gradient descent to find good parameters (contrast to boosting)
  - create more complex adaptive bases by combining simpler bases
    - leads to **deep neural networks**

Adaptive Radial Bases



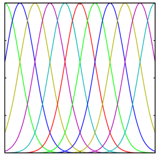
$\phi_d(x) = e^{-\frac{(x-\mu_d)^2}{s^2}}$

Gaussian bases, or radial bases

non-adaptive case

model: $f(x; w) = \sum_d w_d \phi_d(x)$

Adaptive Radial Bases



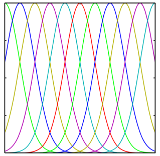
$\phi_d(x) = e^{-\frac{(x-\mu_d)^2}{s^2}}$

Gaussian bases, or radial bases

non-adaptive case

model: $f(x; w) = \sum_d w_d \phi_d(x)$

cost: $J(w) = \frac{1}{2} \sum_n (f(x^{(n)}; w) - y^{(n)})^2$



$\phi_d(x) = e^{-\frac{(x-\mu_d)^2}{s^2}}$

Adaptive Radial Bases

Gaussian bases, or radial bases

non-adaptive case

model: $f(x; w) = \sum_d w_d \phi_d(x)$

cost: $J(w) = \frac{1}{2}\sum_n (f(x^{(n)}; w) - y^{(n)})^2$

the center are fixed

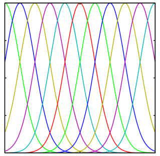
```
#x: N
#y: N
plt.plot(x, y, 'b.')
phi = lambda x,mu: np.exp(-(x-mu)**2)
mu = np.linspace(0,4,10) #4 Gaussians bases
Phi = phi(x[:,None], mu[None,:]) #N x 10
w = np.linalg.lstsq(Phi, y)[0]
yh = np.dot(Phi,w)
plt.plot(x, yh, 'g-')
```

the model is linear in its parameters

the cost is convex in w (unique minimum)

even has a closed form solution

Adaptive Radial Bases



$\phi_d(x) = e^{-\frac{(x-\mu_d)^2}{s^2}}$

Gaussian bases, or radial bases

non-adaptive case

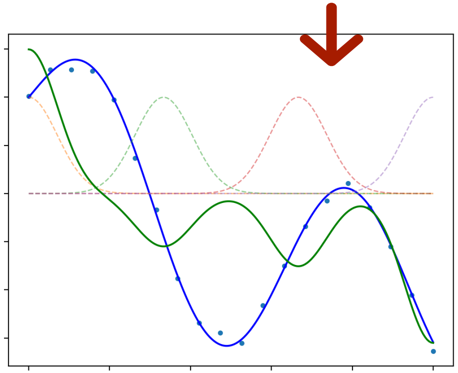
model: $f(x;w) = \sum_d w_d \phi_d(x)$

cost: $J(w) = \frac{1}{2}\sum_n (f(x^{(n)};w) - y^{(n)})^2$

the model is linear in its parameters

the cost is convex in w (unique minimum)

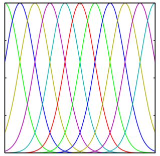
even has a closed form solution



the center are fixed

```
#x: N
#y: N
plt.plot(x, y, 'b.')
phi = lambda x,mu: np.exp(-(x-mu)**2)
mu = np.linspace(0,4,10) #4 Gaussians bases
Phi = phi(x[:,None], mu[None,:]) #N x 10
w = np.linalg.lstsq(Phi, y)[0]
yh = np.dot(Phi,w)
plt.plot(x, yh, 'g-')
```

Adaptive Radial Bases



$$\phi_d(x) = e^{-\frac{(x-\mu_d)^2}{s^2}}$$

Gaussian bases, or radial bases

non-adaptive case

model: $f(x;w) = \sum_d w_d \phi_d(x)$

cost: $J(w) = \frac{1}{2} \sum_n (f(x^{(n)};w) - y^{(n)})^2$

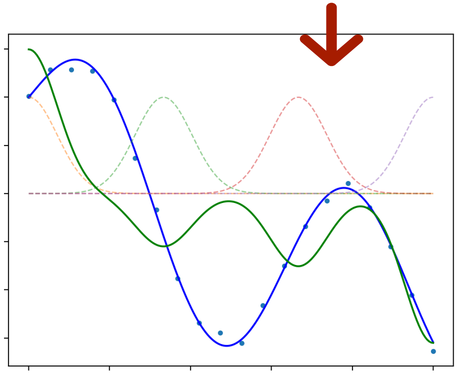
the center are fixed

```
#x: N
#y: N
plt.plot(x, y, 'b.')
phi = lambda x,mu: np.exp(-(x-mu)**2)
mu = np.linspace(0,4,10) #4 Gaussians bases
Phi = phi(x[:,None], mu[None,:]) #N x 10
w = np.linalg.lstsq(Phi, y)[0]
yh = np.dot(Phi,w)
plt.plot(x, yh, 'g-')
```

the model is linear in its parameters

the cost is convex in w (unique minimum)

even has a closed form solution



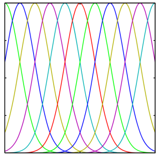
adaptive case

we can make the bases adaptive by learning these centers

model: $f(x;w,\mu) = \sum_d w_d \phi_d(x;\mu_d)$

how to minimize the cost?

Adaptive Radial Bases



$\phi_d(x) = e^{-\frac{(x-\mu_d)^2}{s^2}}$

Gaussian bases, or radial bases

non-adaptive case

model: $f(x; w) = \sum_d w_d \phi_d(x)$

cost: $J(w) = \frac{1}{2} \sum_n (f(x^{(n)}; w) - y^{(n)})^2$

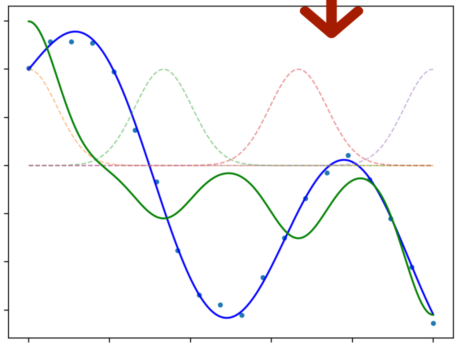
the center are fixed

```python
#x: N
#y: N
plt.plot(x, y, 'b.')
phi = lambda x,mu: np.exp(-(x-mu)**2)
mu = np.linspace(0,4,10) #4 Gaussians bases
Phi = phi(x[:,None], mu[None,:]) #N x 10
w = np.linalg.lstsq(Phi, y)[0]
yh = np.dot(Phi,w)
plt.plot(x, yh, 'g-')
```

the model is linear in its parameters

the cost is convex in w (unique minimum)

even has a closed form solution



adaptive case

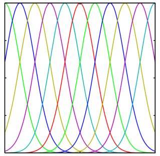
we can make the bases adaptive by learning these centers

model: $f(x; w, \mu) = \sum_d w_d \phi_d(x; \mu_d)$

how to minimize the cost?

not convex in all model parameters

use gradient descent to find a local minimum



$\phi_d(x) = e^{-\frac{(x-\mu_d)^2}{s^2}}$

Gaussian bases, or radial bases

Adaptive Radial Bases

non-adaptive case

model: $f(x; w) = \sum_d w_d \phi_d(x)$

cost: $J(w) = \frac{1}{2} \sum_n (f(x^{(n)}; w) - y^{(n)})^2$

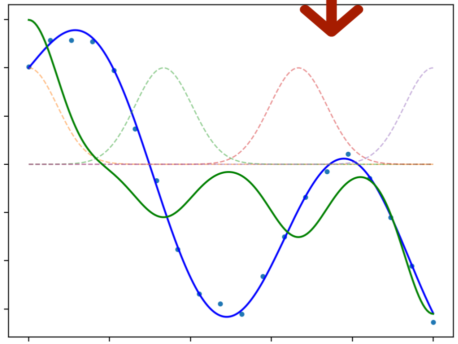
the center are fixed

```
#x: N
#y: N
plt.plot(x, y, 'b.')
phi = lambda x,mu: np.exp(-(x-mu)**2)
mu = np.linspace(0,4,10) #4 Gaussians bases
Phi = phi(x[:,None], mu[None,:]) #N x 10
w = np.linalg.lstsq(Phi, y)[0]
yh = np.dot(Phi,w)
plt.plot(x, yh, 'g-')
```

the model is linear in its parameters

the cost is convex in w (unique minimum)

even has a closed form solution



adaptive case

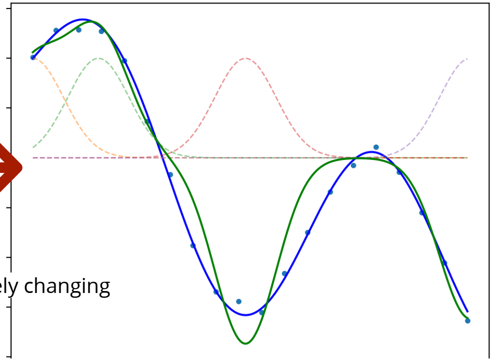
we can make the bases adaptive by learning these centers

model: $f(x; w, \mu) = \sum_d w_d \phi_d(x; \mu_d)$

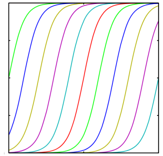
how to minimize the cost?

not convex in all model parameters

use gradient descent to find a local minimum



note that the basis centers are adaptively changing



$$\phi_d(x) = \frac{1}{1+e^{-(\frac{x-\mu_d}{s_d})}}$$

Sigmoid Bases

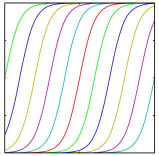
using adaptive sigmoid bases gives us a neural network

non-adaptive case

μ_d

$s_d = 1$

Sigmoid Bases



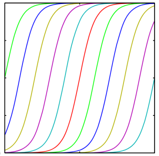
$$\phi_d(x) = \frac{1}{1+e^{-(\frac{x-\mu_d}{s_d})}}$$

using adaptive sigmoid bases gives us a neural network

non-adaptive case

- μ_d is fixed to D locations
- $s_d = 1$

Sigmoid Bases



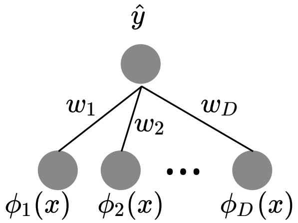
$$\phi_d(x) = \frac{1}{1+e^{-\left(\frac{x-\mu_d}{s_d}\right)}}$$

using adaptive sigmoid bases gives us a neural network

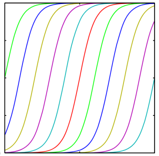
non-adaptive case

- μ_d is fixed to D locations
- $s_d = 1$

model: $f(x; w) = \sum_d w_d \phi_d(x)$



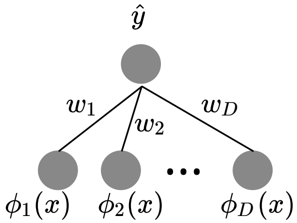

```
#x: N
#y: N
plt.plot(x, y, 'b.')
phi = lambda x,mu,sigma: 1/(1 + np.exp(-(x - mu)))
mu = np.linspace(0,3,10)
Phi = phi(x[:,None], mu[None,:]) #N x 10
w = np.linalg.lstsq(Phi, y)[0]
yh = np.dot(Phi,w)
plt.plot(x, yh, 'g-')
```



$$\phi_d(x) = \frac{1}{1+e^{-\left(\frac{x-\mu_d}{s_d}\right)}}$$

Sigmoid Bases

using adaptive sigmoid bases gives us a neural network

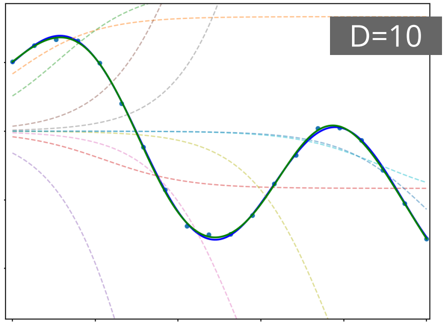


non-adaptive case

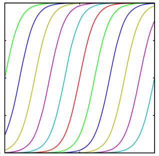
- μ_d is fixed to D locations
- $s_d = 1$

model: $f(x; w) = \sum_d w_d \phi_d(x)$

```
#x: N
#y: N
plt.plot(x, y, 'b.')
phi = lambda x,mu,sigma: 1/(1 + np.exp(-(x - mu)))
mu = np.linspace(0,3,10)
Phi = phi(x[:,None], mu[None,:]) #N x 10
w = np.linalg.lstsq(Phi, y)[0]
yh = np.dot(Phi,w)
plt.plot(x, yh, 'g-')
```

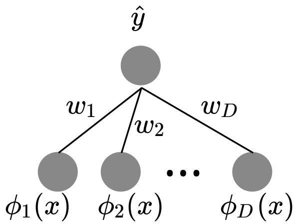


Sigmoid Bases



$$\phi_d(x) = \frac{1}{1+e^{-\left(\frac{x-\mu_d}{s_d}\right)}}$$

using adaptive sigmoid bases gives us a neural network

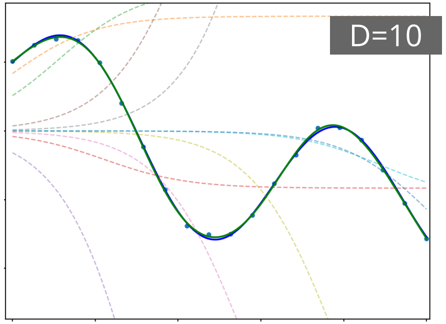


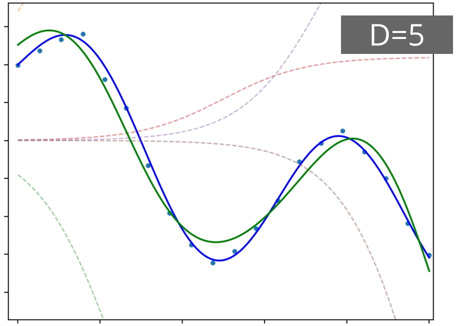
non-adaptive case

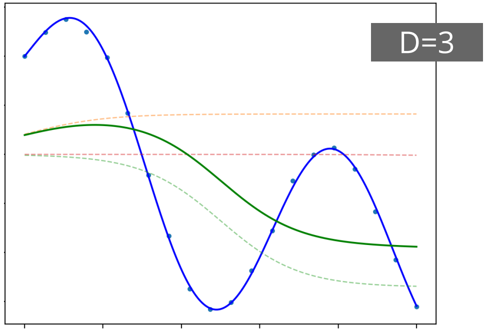
- μ_d is fixed to D locations
- $s_d = 1$

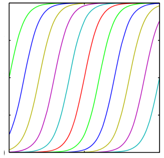
model: $f(x; w) = \sum_d w_d \phi_d(x)$

```
#x: N
#y: N
plt.plot(x, y, 'b.')
phi = lambda x,mu,sigma: 1/(1 + np.exp(-(x - mu)))
mu = np.linspace(0,3,10)
Phi = phi(x[:,None], mu[None,:]) #N x 10
w = np.linalg.lstsq(Phi, y)[0]
yh = np.dot(Phi,w)
plt.plot(x, yh, 'g-')
```









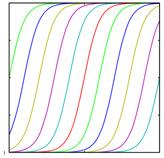
$\phi_d(x) = \frac{1}{1+e^{-(\frac{x-\mu_d}{s_d})}}$

Adaptive Sigmoid Bases

rewrite the sigmoid basis

$$\phi_d(x) = \sigma(\frac{x-\mu_d}{s_d}) = \sigma(v_d x + b_d)$$

Adaptive Sigmoid Bases



$\phi_d(x) = \frac{1}{1+e^{-(\frac{x-\mu_d}{s_d})}}$

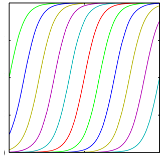
rewrite the sigmoid basis

$$\phi_d(x) = \sigma(\frac{x-\mu_d}{s_d}) = \sigma(v_d x + b_d)$$

each basis is the logistic regression model $\phi_d(x) = \sigma(v_d^\top x + b_d)$

assuming input is higher than one dimension

Adaptive Sigmoid Bases



$\phi_d(x) = \frac{1}{1+e^{-(\frac{x-\mu_d}{s_d})}}$

rewrite the sigmoid basis

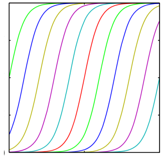
$$\phi_d(x) = \sigma(\frac{x-\mu_d}{s_d}) = \sigma(v_d x + b_d)$$

each basis is the logistic regression model $\phi_d(x) = \sigma(v_d^\top x + b_d)$

assuming input is higher than one dimension

model: $f(x; w, v, b) = \sum_d w_d \sigma(v_d x + b_d)$

Adaptive Sigmoid Bases



$\phi_d(x) = \frac{1}{1+e^{-(\frac{x-\mu_d}{s_d})}}$

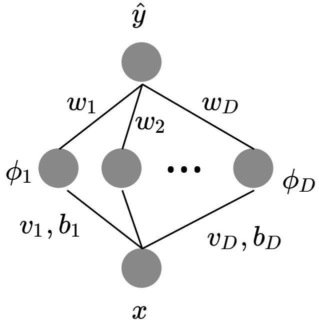
rewrite the sigmoid basis

$$\phi_d(x) = \sigma(\frac{x-\mu_d}{s_d}) = \sigma(v_d x + b_d)$$

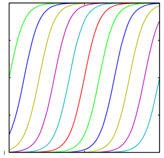
each basis is the logistic regression model $\phi_d(x) = \sigma(v_d^\top x + b_d)$

assuming input is higher than one dimension

model: $f(x; w, v, b) = \sum_d w_d \sigma(v_d x + b_d)$ this is a **neural network** with two layers →



Adaptive Sigmoid Bases



$\phi_d(x) = \frac{1}{1+e^{-(\frac{x-\mu_d}{s_d})}}$

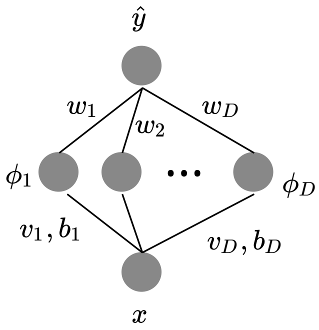
rewrite the sigmoid basis

$$\phi_d(x) = \sigma(\frac{x-\mu_d}{s_d}) = \sigma(v_d x + b_d)$$

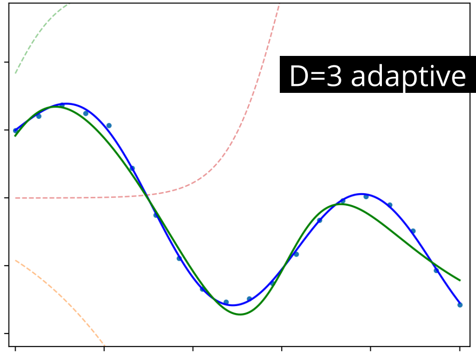
each basis is the logistic regression model $\phi_d(x) = \sigma(v_d^\top x + b_d)$

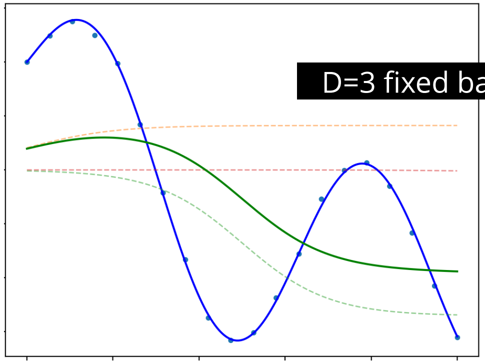
assuming input is higher than one dimension

model: $f(x; w, v, b) = \sum_d w_d \sigma(v_d x + b_d)$ this is a **neural network** with two layers



optimize using gradient descent (find a local optima)





Multilayer Perceptron (MLP)

suppose we have

- D inputs $x_1, \ldots, x_D$
- K outputs $\hat{y}_1, \ldots, \hat{y}_K$
- M hidden *units* $z_1, \ldots, z_M$

model

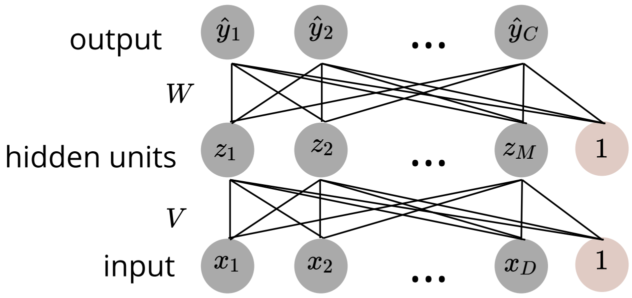
$$\hat{y}_k = g\left(\sum_m W_{k,m} h\left(\sum_d V_{m,d} x_d\right)\right)$$

nonlinearity, activation function: we have different choices

more compressed form

$$\hat{y} = g\big(W\, h(V\, x)\big)$$

non-linearities are applied elementwise

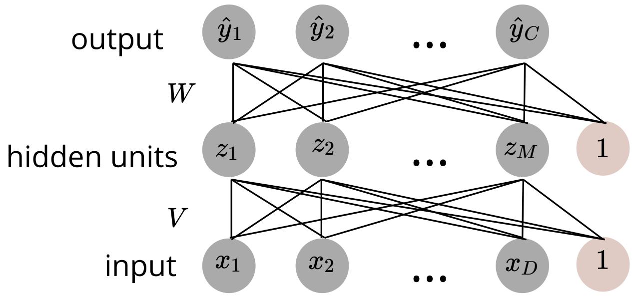


for simplicity we may drop bias terms

Regression using Neural Networks

the choice of **activation function** in the final layer depends on the task

model $\hat{y} = g\big(W\, h(V\, x)\big)$



Regression using Neural Networks

the choice of **activation function** in the final layer depends on the task

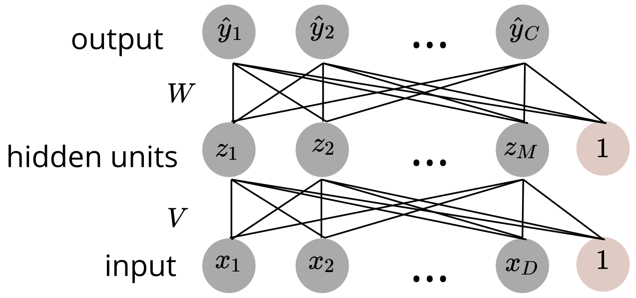
model $\hat{y} = g\big(W\, h(V\, x)\big)$

regression $\hat{y} = g(\, W z\,) = W z$

we may have one or more output variables

identity function + L2 loss : Gaussian likelihood

$L(y, \hat{y}) = \frac{1}{2}||y - \hat{y}||_2^2 = \log \mathcal{N}(y; \hat{y}, \beta \mathbf{I}) + \text{constant}$



Regression using Neural Networks

the choice of **activation function** in the final layer depends on the task

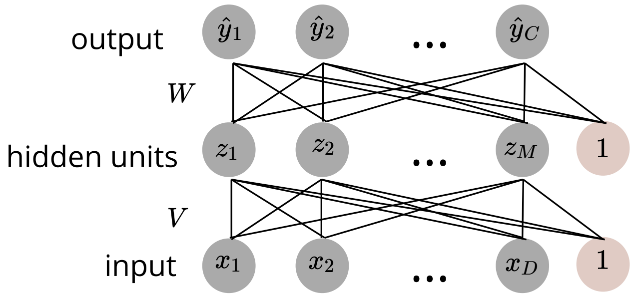
model $\hat{y} = g\big(W\, h(V\, x)\big)$

regression $\hat{y} = g(\, W z\,) = W z$

we may have one or more output variables

identity function + L2 loss : Gaussian likelihood

$$L(y, \hat{y}) = \tfrac{1}{2}||y - \hat{y}||_2^2 = \log \mathcal{N}(y; \hat{y}, \beta\mathbf{I}) + \text{constant}$$



more generally

we may explicitly produce a distribution at output - *e.g.,*

- mean and variance of a Gaussian
- mixture of Gaussians

the loss will be the log-likelihood of the data under our model

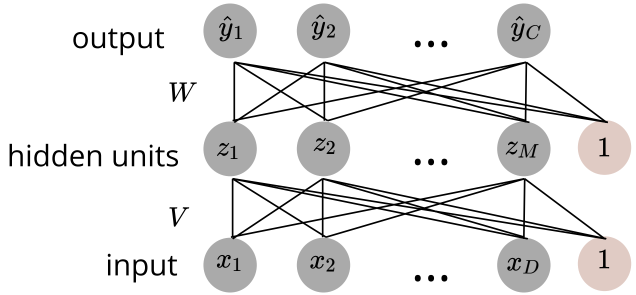
$$L(y, \hat{y}) = \log p(y; f(x))$$

neural network outputs the parameters of a distribution

Classification using neural networks

the choice of activation function in the final layer depends on the task

model $\hat{y} = g\big(W\, h(V\, x)\big)$



Classification using neural networks

the choice of activation function in the final layer depends on the task

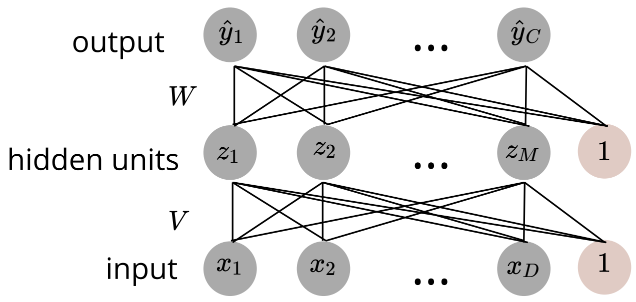
model $\hat{y} = g\big(W\,h(V\,x)\big)$

binary classification $\hat{y} = g(Wz) = \big(1 + e^{-Wz}\big)^{-1}$

scalar output C=1

logistic sigmoid + CE loss: Bernouli likelihood

$$L(y, \hat{y}) = y \log \hat{y} + (1-y)\log(1-\hat{y}) = \log \mathrm{Bernouli}(y; \hat{y})$$



Classification using neural networks

the choice of activation function in the final layer depends on the task

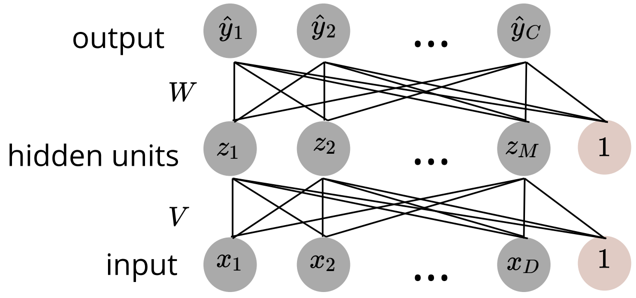
model $\hat{y} = g\big(W\, h(V\, x)\big)$

binary classification $\hat{y} = g(Wz) = \big(1 + e^{-Wz}\big)^{-1}$

scalar output C=1

logistic sigmoid + CE loss: Bernouli likelihood

$L(y, \hat{y}) = y \log \hat{y} + (1 - y) \log(1 - \hat{y}) = \log \mathrm{Bernouli}(y; \hat{y})$



multiclass classification $\hat{y} = g(Wz) = \mathrm{softmax}(Wz)$

C is the number of classes

softmax + multi-class CE loss: categorical likelihood

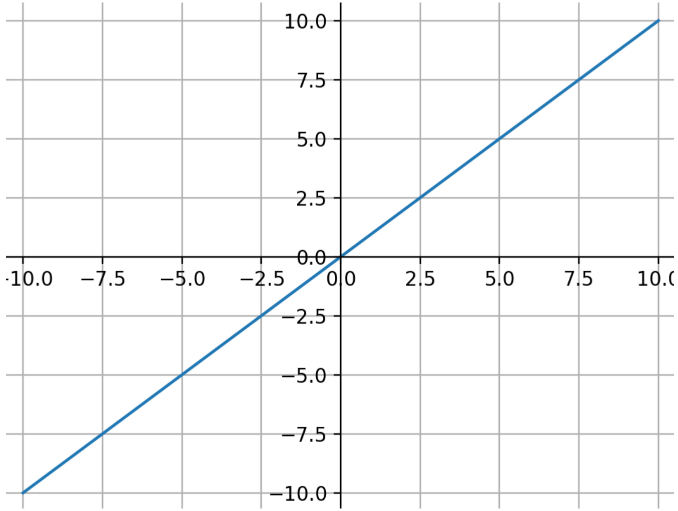
$L(y, \hat{y}) = \sum_k y_k \log \hat{y}_k = \log \mathrm{Categorical}(y; \hat{y})$

Activation function

for **middle layer(s)** there is more freedom in the choice of activation function

Activation function

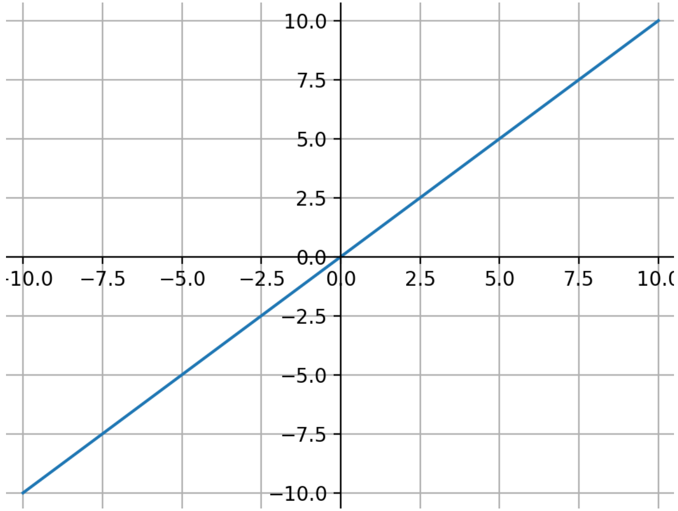
for **middle layer(s)** there is more freedom in the choice of activation function



$h(x) = x$ **identity** (no activation function)

Activation function

for **middle layer(s)** there is more freedom in the choice of activation function



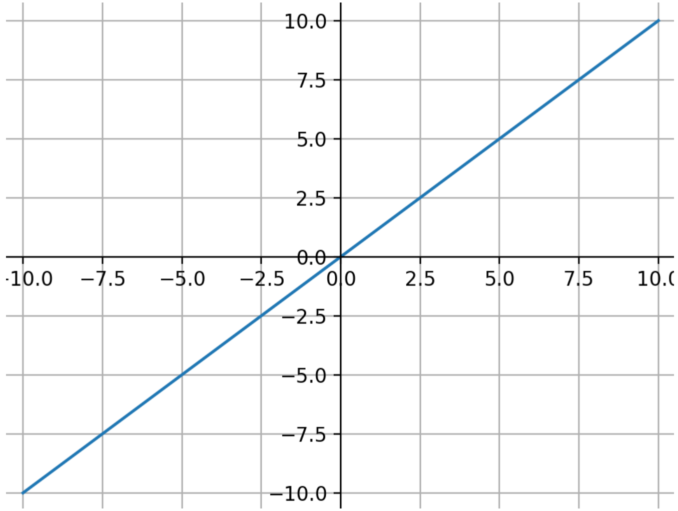
$h(x) = x$ **identity** (no activation function)

composition of two linear functions is linear

$$\underbrace{\overset{K \times M}{W}\,\overset{M \times D}{V}}_{W'} x = \overset{K \times D}{W'} x$$

Activation function

for **middle layer(s)** there is more freedom in the choice of activation function



$h(x) = x$ **identity** (no activation function)

composition of two linear functions is linear

$$\underbrace{\overset{K \times M}{W}\overset{M \times D}{V}}_{W'} x = \overset{K \times D}{W'} x$$

so nothing is gained (in representation power) by stacking linear layers

exception: if $M < \min(D, K)$ then the hidden layer is compressing the data (W' is low-rank)

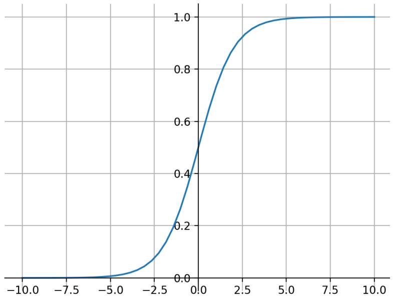
this idea is used in dimensionality reduction (later!)

Activation function

for **middle layer(s)** there is more freedom in the choice of activation function

Activation function

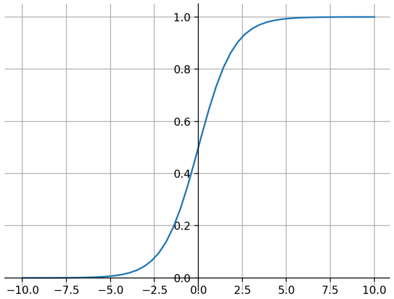
for **middle layer(s)** there is more freedom in the choice of activation function



$h(x) = \sigma(x) = \frac{1}{1+e^{-x}}$ **logistic function**

Activation function

for **middle layer(s)** there is more freedom in the choice of activation function



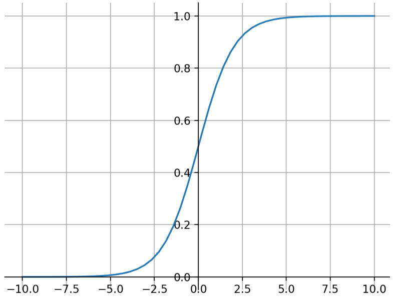
$h(x) = \sigma(x) = \frac{1}{1+e^{-x}}$ **logistic function**

the same function used in logistic regression

used to be the function of choice in neural networks

Activation function

for **middle layer(s)** there is more freedom in the choice of activation function



$h(x) = \sigma(x) = \frac{1}{1+e^{-x}}$ **logistic function**

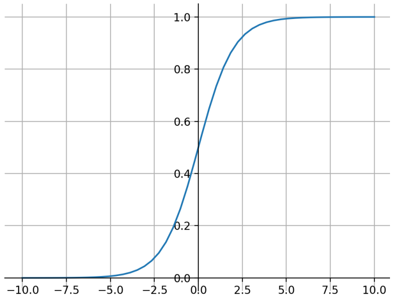
the same function used in logistic regression

used to be the function of choice in neural networks

away from zero it changes slowly, so the derivative is small (leads to vanishing gradient)

Activation function

for **middle layer(s)** there is more freedom in the choice of activation function



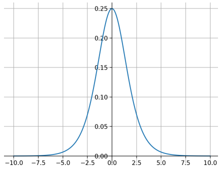
$h(x) = \sigma(x) = \frac{1}{1+e^{-x}}$ **logistic function**

the same function used in logistic regression

used to be the function of choice in neural networks

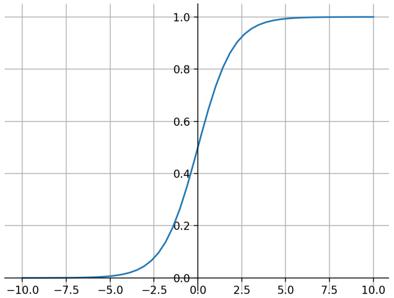
away from zero it changes slowly, so the derivative is small (leads to vanishing gradient)

its derivative is easy to remember $\frac{\partial}{\partial x}\sigma(x) = \sigma(x)(1 - \sigma(x))$



Activation function

for **middle layer(s)** there is more freedom in the choice of activation function



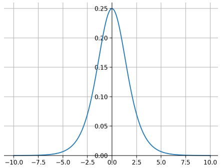
$h(x) = \sigma(x) = \frac{1}{1+e^{-x}}$ **logistic function**

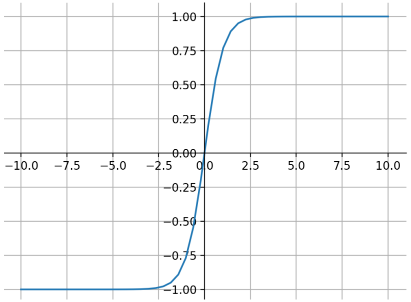
the same function used in logistic regression

used to be the function of choice in neural networks

away from zero it changes slowly, so the derivative is small (leads to vanishing gradient)

its derivative is easy to remember $\frac{\partial}{\partial x}\sigma(x) = \sigma(x)(1-\sigma(x))$



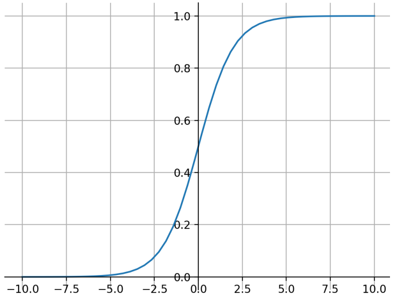


$h(x) = 2\sigma(x) - 1 = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ **hyperbolic tangent**

similar to sigmoid, but symmetric

Activation function

for **middle layer(s)** there is more freedom in the choice of activation function



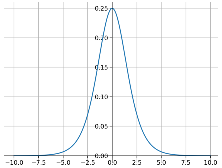
$h(x) = \sigma(x) = \frac{1}{1+e^{-x}}$ **logistic function**

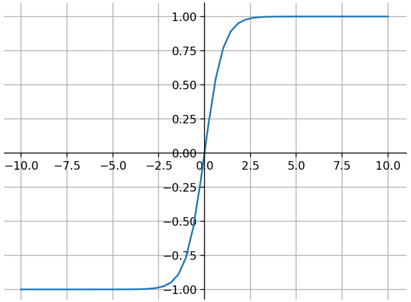
the same function used in logistic regression

used to be the function of choice in neural networks

away from zero it changes slowly, so the derivative is small (leads to vanishing gradient)

its derivative is easy to remember $\frac{\partial}{\partial x}\sigma(x) = \sigma(x)(1-\sigma(x))$





$h(x) = 2\sigma(x) - 1 = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ **hyperbolic tangent**

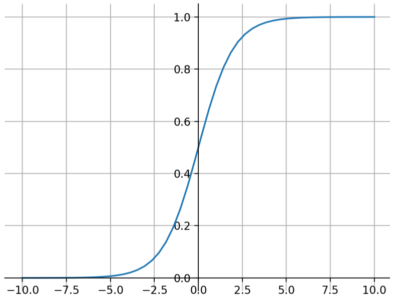
similar to sigmoid, but symmetric

often better for optimization because close to zero it similar to a linear function

(rather than an affine function when using logistic)

Activation function

for **middle layer(s)** there is more freedom in the choice of activation function



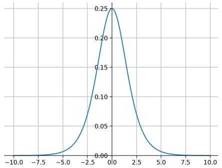
$h(x) = \sigma(x) = \frac{1}{1+e^{-x}}$ **logistic function**

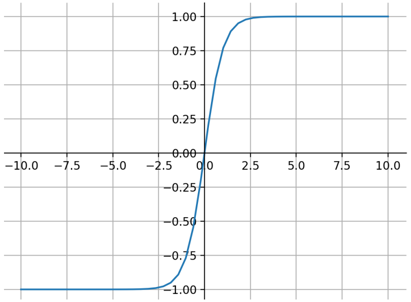
the same function used in logistic regression

used to be the function of choice in neural networks

away from zero it changes slowly, so the derivative is small (leads to vanishing gradient)

its derivative is easy to remember $\frac{\partial}{\partial x}\sigma(x) = \sigma(x)(1-\sigma(x))$





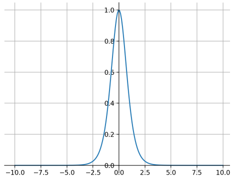
$h(x) = 2\sigma(x) - 1 = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ **hyperbolic tangent**

similar to sigmoid, but symmetric

often better for optimization because close to zero it similar to a linear function

(rather than an affine function when using logistic)

similar problem with vanishing gradient



$\frac{\partial}{\partial x}\tanh(x) = 1 - \tanh(x)^2$

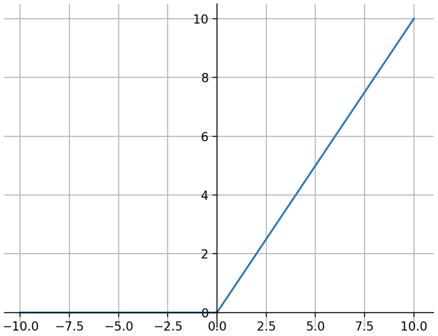
Activation function

for **middle layer(s)** there is more freedom in the choice of activation function

γ

Activation function

for **middle layer(s)** there is more freedom in the choice of activation function

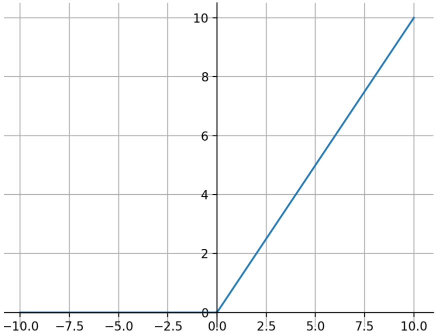


$h(x) = \max(0, x)$ Rectified Linear Unit (**ReLU**)

γ

Activation function

for **middle layer(s)** there is more freedom in the choice of activation function



$h(x) = \max(0, x)$ Rectified Linear Unit (**ReLU**)

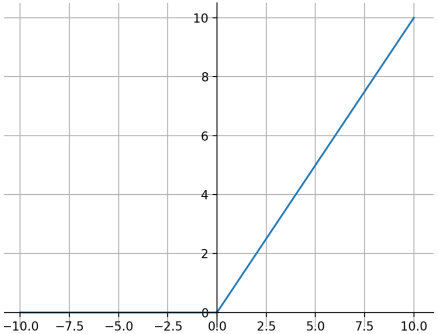
replacing logistic with ReLU significantly improves the training of deep networks

zero derivative if the unit is "inactive"
initialization should ensure active units at the beginning of optimization

γ

Activation function

for **middle layer(s)** there is more freedom in the choice of activation function

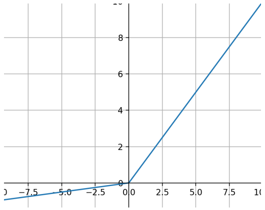


$h(x) = \max(0, x)$ Rectified Linear Unit (**ReLU**)

replacing logistic with ReLU significantly improves the training of deep networks

zero derivative if the unit is "inactive"
initialization should ensure active units at the beginning of optimization

leaky ReLU $h(x) = \max(0, x) + \gamma \min(0, x)$

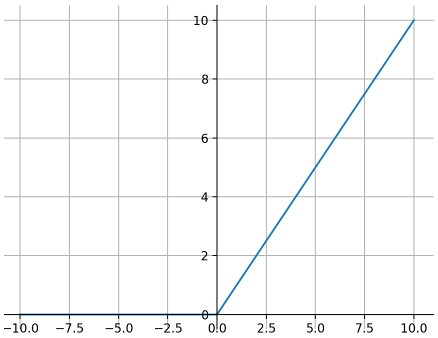


fixes the zero-gradient problem

γ

Activation function

for **middle layer(s)** there is more freedom in the choice of activation function

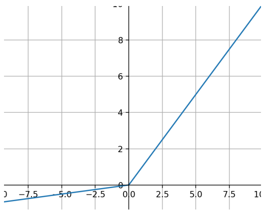


$h(x) = \max(0, x)$ Rectified Linear Unit (**ReLU**)

replacing logistic with ReLU significantly improves the training of deep networks

zero derivative if the unit is "inactive"
initialization should ensure active units at the beginning of optimization

leaky ReLU $h(x) = \max(0, x) + \gamma \min(0, x)$

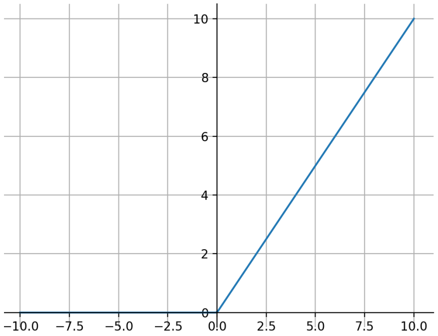


fixes the zero-gradient problem

parameteric ReLU:
make γ a learnable parameter

Activation function

for **middle layer(s)** there is more freedom in the choice of activation function

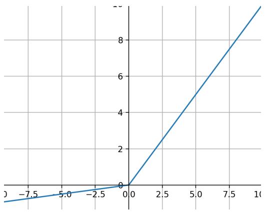


$h(x) = \max(0, x)$ Rectified Linear Unit (**ReLU**)

replacing logistic with ReLU significantly improves the training of deep networks

zero derivative if the unit is "inactive"
initialization should ensure active units at the beginning of optimization

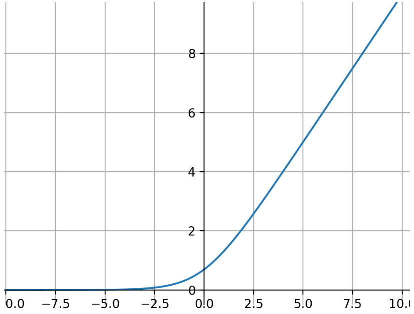
leaky ReLU $h(x) = \max(0, x) + \gamma \min(0, x)$



fixes the zero-gradient problem

parameteric ReLU:
make γ a learnable parameter

Softplus (differentiable everywhere) $h(x) = \log(1 + e^{x})$



it doesn't perform as well in practice

Network architecture

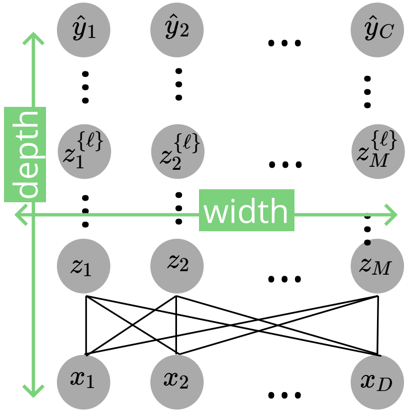
architecture is the overall structure of the network

Network architecture

architecture is the overall structure of the network

feedforward network (aka multilayer perceptron)

- can have many layers
- # layers is called the **depth** of the network

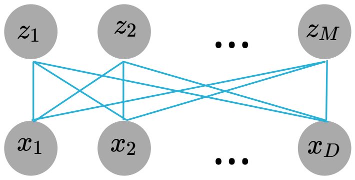


Network architecture

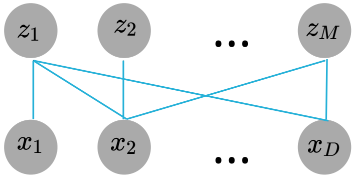
architecture is the overall structure of the network

feedforward network (aka multilayer perceptron)

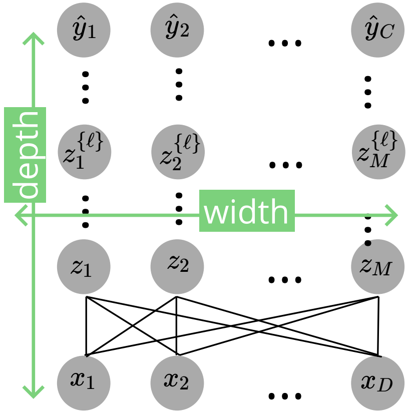
- can have many layers
- # layers is called the **depth** of the network
- each layer can be **fully connected** (dense) or sparse



fully connected



sparsely connected

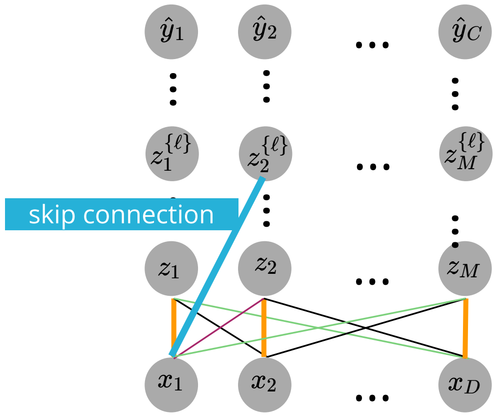


Network architecture

architecture is the overall structure of the network

feed-forward network (aka multilayer perceptron**)**

- can have many layers
- # layers is called the **depth** of the network
- each layer can be **fully connected** (dense) or sparse

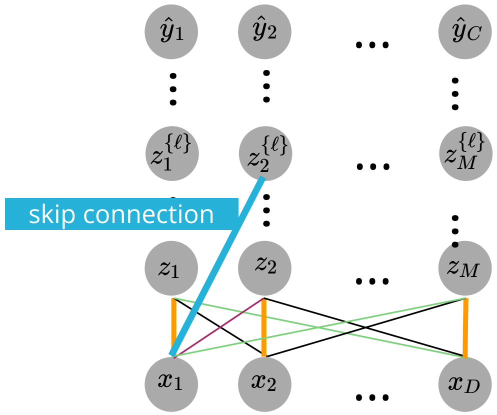


Network architecture

architecture is the overall structure of the network

feed-forward network (aka multilayer perceptron**)**

- can have many layers
- # layers is called the **depth** of the network
- each layer can be **fully connected** (dense) or sparse
- layers may have **skip layer connections**
  - helps with gradient flow

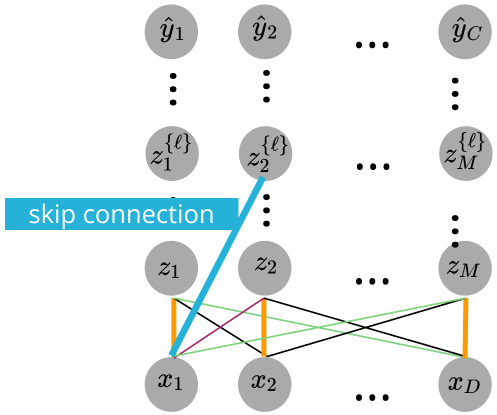


Network architecture

architecture is the overall structure of the network

feed-forward network (aka multilayer perceptron**)**

- can have many layers
- # layers is called the **depth** of the network
- each layer can be **fully connected** (dense) or sparse
- layers may have **skip layer connections**
  - helps with gradient flow
- units may have different **activations**

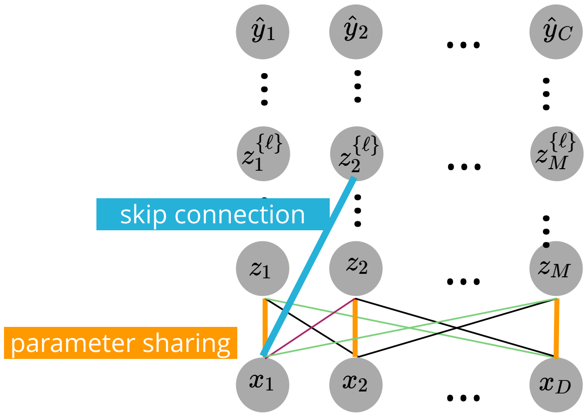


Network architecture

architecture is the overall structure of the network

feed-forward network (aka multilayer perceptron**)**

- can have many layers
- # layers is called the **depth** of the network
- each layer can be **fully connected** (dense) or sparse
- layers may have **skip layer connections**
  - helps with gradient flow
- units may have different **activations**
- parameters may be shared across units (*e.g.*, in conv-nets)

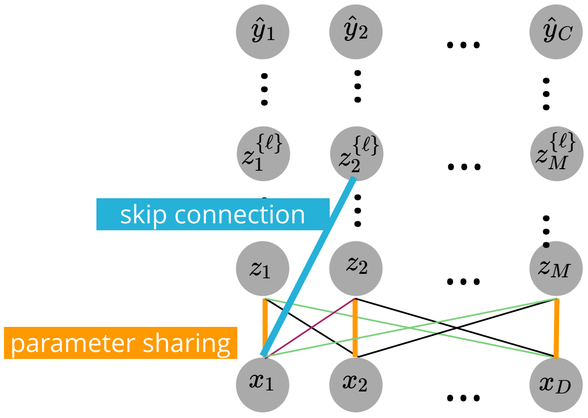


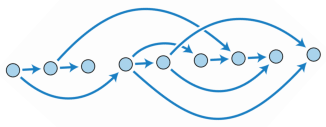
Network architecture

architecture is the overall structure of the network

feed-forward network (aka multilayer perceptron)

- can have many layers
- # layers is called the **depth** of the network
- each layer can be **fully connected** (dense) or sparse
- layers may have **skip layer connections**
  - helps with gradient flow
- units may have different **activations**
- parameters may be shared across units (*e.g.*, in conv-nets)





more generally a directed acyclic graph (DAG) expresses the feed-forward architecture

Expressive power

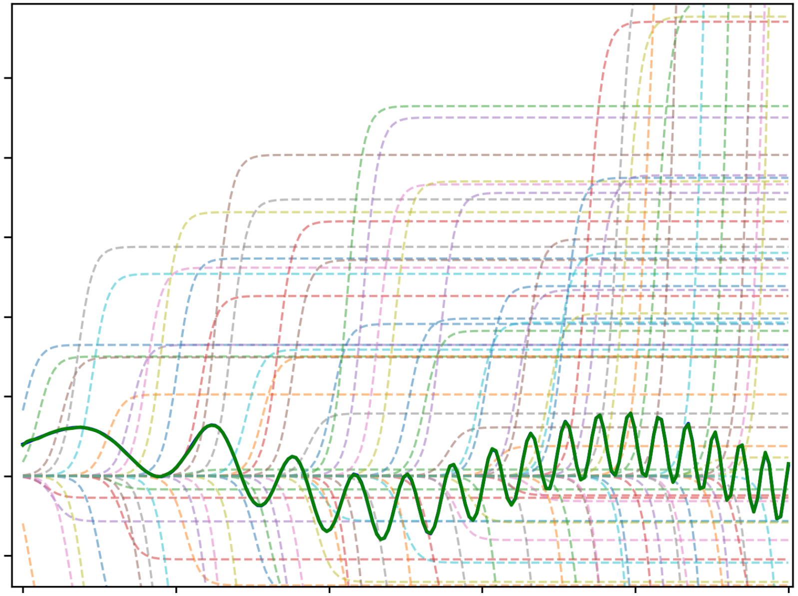
universal approximation theorem

an MLP with single hidden layer can approximate any continuous function with arbitrary accuracy

Expressive power

universal approximation theorem

an MLP with single hidden layer can approximate any continuous function with arbitrary accuracy



for 1D input we can see this even with **fixed bases**

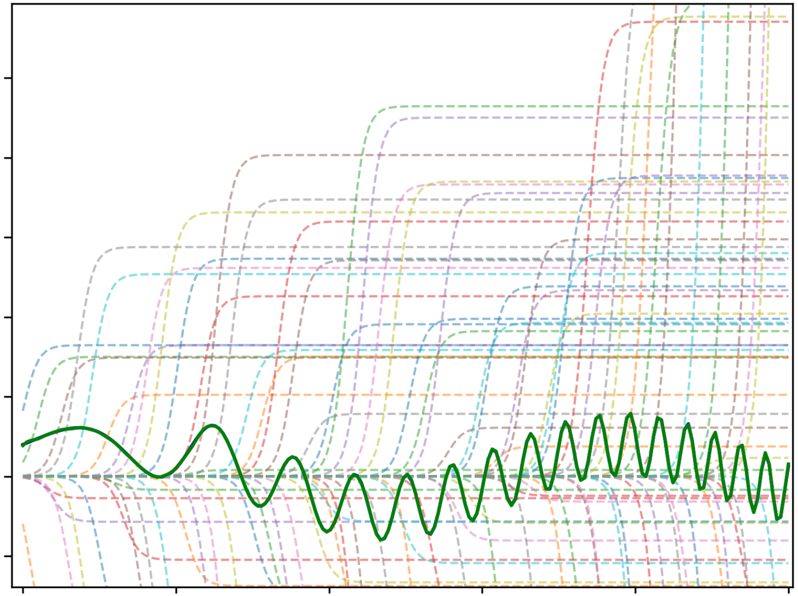
M = 100 in this example

the fit is good (hard to see the blue line)

Expressive power

universal approximation theorem

an MLP with single hidden layer can approximate any continuous function with arbitrary accuracy



for 1D input we can see this even with **fixed bases**

M = 100 in this example

the fit is good (hard to see the blue line)

however # bases (M) should grow exponentially with D (**curse of dimensionality**)

Depth vs Width

universal approximation theorem

an MLP with single hidden layer can approximate any continuous function on with arbitrary accuracy

Caveats

- we may need a very wide network (large M)
- this is only about training error, we care about test error

Depth vs Width

universal approximation theorem

an MLP with single hidden layer can approximate any continuous function on with arbitrary accuracy

Caveats

- we may need a very wide network (large M)
- this is only about training error, we care about test error

Deep networks (with ReLU activation) of bounded width are also shown to be universal

Depth vs Width

universal approximation theorem

an MLP with single hidden layer can approximate any continuous function on with arbitrary accuracy

Caveats

- we may need a very wide network (large M)
- this is only about training error, we care about test error

Deep networks (with ReLU activation) of bounded width are also shown to be universal

empirically it is observed that increasing depth is often more effective than increasing width (#parameters per layer)

assuming a compositional functional form (through depth) is a useful inductive bias

Depth vs Width

universal approximation theorem

an MLP with single hidden layer can approximate any continuous function on with arbitrary accuracy

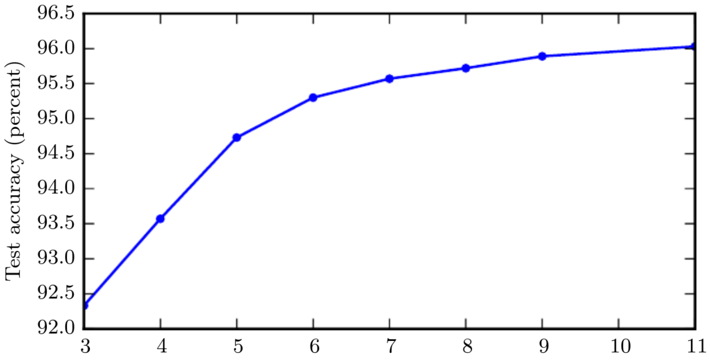
Caveats

- we may need a very wide network (large M)
- this is only about training error, we care about test error

Deep networks (with ReLU activation) of bounded width are also shown to be universal

empirically it is observed that increasing depth is often more effective than increasing width (#parameters per layer)

assuming a compositional functional form (through depth) is a useful inductive bias



Depth vs Width

universal approximation theorem

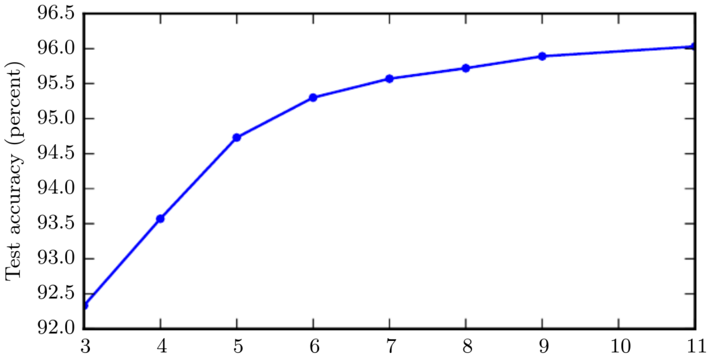
an MLP with single hidden layer can approximate any continuous function on with arbitrary accuracy

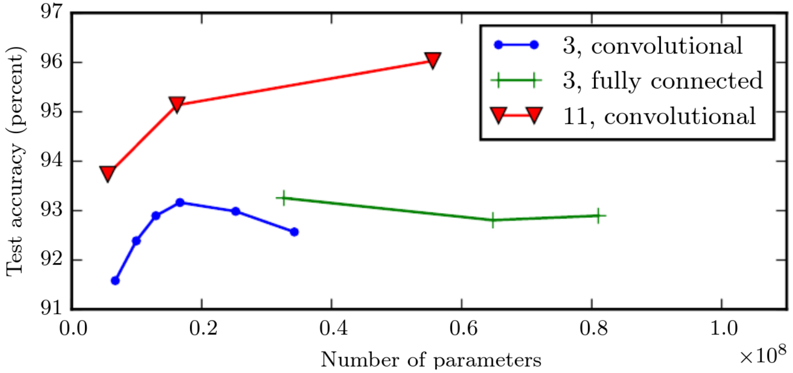
Caveats

- we may need a very wide network (large M)
- this is only about training error, we care about test error

Deep networks (with ReLU activation) of bounded width are also shown to be universal

empirically it is observed that increasing depth is often more effective than increasing width (#parameters per layer)
assuming a compositional functional form (through depth) is a useful inductive bias





Depth vs Width

universal approximation theorem

an MLP with single hidden layer can approximate any continuous function on with arbitrary accuracy

Caveats

- we may need a very wide network (large M)
- this is only about training error, we care about test error

Deep networks (with ReLU activation) of bounded width are also shown to be universal

Depth vs Width

universal approximation theorem

an MLP with single hidden layer can approximate any continuous function on with arbitrary accuracy

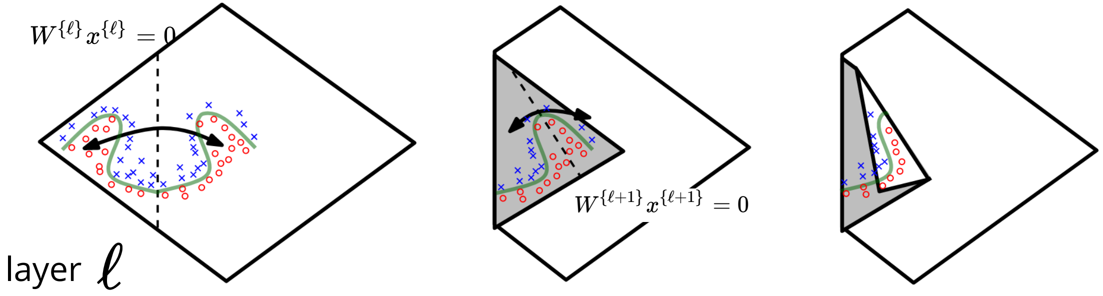
Caveats

- we may need a very wide network (large M)
- this is only about training error, we care about test error

Deep networks (with ReLU activation) of bounded width are also shown to be universal

number of regions (in which the network is linear) grows exponentially with depth

simplified demonstration $h(W^{\{\ell\}}x) = |W^{\{\ell\}}x|$



Regularization strategies

universality of neural networks also means they can overfit

strategies for variance reduction:

Regularization strategies

universality of neural networks also means they can overfit

strategies for variance reduction:

- L1 and L2 regularization *(weight decay)*

Regularization strategies

universality of neural networks also means they can overfit

strategies for variance reduction:

- L1 and L2 regularization *(weight decay)*
- data augmentation
- noise robustness
- early stopping
- bagging and dropout

Regularization strategies

universality of neural networks also means they can overfit

strategies for variance reduction:

- L1 and L2 regularization *(weight decay)*
- data augmentation
- noise robustness
- early stopping
- bagging and dropout
- sparse representations *(e.g., L1 penalty on hidden unit activations)*

Regularization strategies

universality of neural networks also means they can overfit

strategies for variance reduction:

- L1 and L2 regularization *(weight decay)*
- data augmentation
- noise robustness
- early stopping
- bagging and dropout
- sparse representations *(e.g., L1 penalty on hidden unit activations)*
- semi-supervised and multi-task learning

Regularization strategies

universality of neural networks also means they can overfit

strategies for variance reduction:

- L1 and L2 regularization *(weight decay)*
- data augmentation
- noise robustness
- early stopping
- bagging and dropout
- sparse representations *(e.g., L1 penalty on hidden unit activations)*
- semi-supervised and multi-task learning
- adversarial training

Regularization strategies

universality of neural networks also means they can overfit

strategies for variance reduction:

- L1 and L2 regularization *(weight decay)*
- data augmentation
- noise robustness
- early stopping
- bagging and dropout
- sparse representations *(e.g., L1 penalty on hidden unit activations)*
- semi-supervised and multi-task learning
- adversarial training
- parameter-tying

Data augmentation

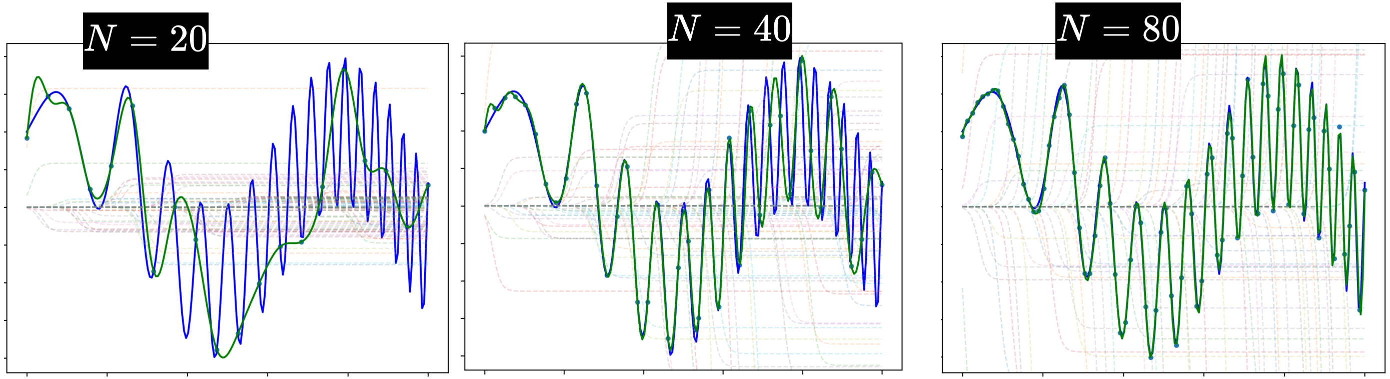
a larger dataset results in a better generalization

Data augmentation

a larger dataset results in a better generalization

example: in all 3 examples below training error is close to zero

however, a larger training dataset leads to better generalization



Data augmentation

a larger dataset results in a better generalization

idea

increase the size of dataset by adding reasonable transformations $\tau(x)$ that change the label in predictable ways; e.g., $f(\tau(x)) = f(x)$

image: https://github.com/aleju/imgaug/blob/master/README.md

Data augmentation

a larger dataset results in a better generalization

idea

increase the size of dataset by adding reasonable transformations $\tau(x)$ that change the label in predictable ways; e.g., $f(\tau(x)) = f(x)$

special approaches to data-augmentation

- adding noise to the input
- adding noise to hidden units
  - noise in higher level of abstraction

image: https://github.com/aleju/imgaug/blob/master/README.md

Data augmentation

a larger dataset results in a better generalization

idea

increase the size of dataset by adding reasonable transformations $\tau(x)$ that change the label in predictable ways; e.g., $f(\tau(x)) = f(x)$

special approaches to data-augmentation

- adding noise to the input
- adding noise to hidden units
  - noise in higher level of abstraction

- learn a **generative model** $\hat{p}(x,y)$ of the data
  - use $x^{(n')}, y^{(n')} \sim \hat{p}$ for training

image: https://github.com/aleju/imgaug/blob/master/README.md

Data augmentation

a larger dataset results in a better generalization

idea

increase the size of dataset by adding reasonable transformations $\tau(x)$
that change the label in predictable ways; e.g., $f(\tau(x)) = f(x)$

special approaches to data-augmentation

- adding noise to the input
- adding noise to hidden units
  - noise in higher level of abstraction

- learn a **generative model** $\hat{p}(x, y)$ of the data
  - use $x^{(n')}, y^{(n')} \sim \hat{p}$ for training

sometimes we an achieve the same goal by designing the models
that are **invariant** to a given set of transformations

image: https://github.com/aleju/imgaug/blob/master/README.md

Noise robustness

make the model robust to noise in

Noise robustness

make the model robust to noise in

- **input** (data augmentation)
- **hidden units** (*e.g.*, in dropout)

Noise robustness

make the model robust to noise in

- **input** (data augmentation)
- **hidden units** (*e.g.*, in dropout)
- **weights** the loss is not sensitive to small changes in the weight (flat minima)

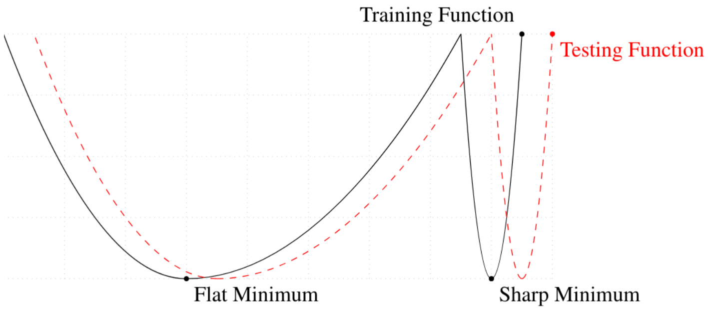


image credit: Keshkar et al'17

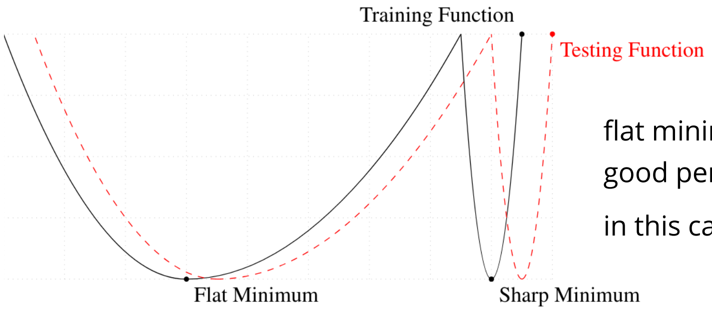
Noise robustness

make the model robust to noise in

input (data augmentation)

hidden units (*e.g.*, in dropout)

weights the loss is not sensitive to small changes in the weight (flat minima)



flat minima generalize better

good performance of SGD using small minibatch is attributed to flat minima

in this case, SGD regularizes the model due to **gradient noise**

image credit: Keshkar et al'17

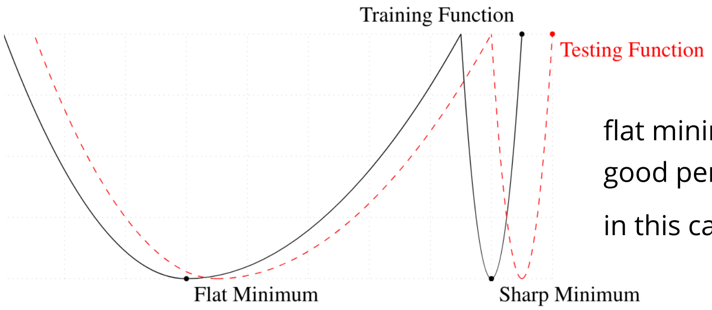
Noise robustness

make the model robust to noise in

input (data augmentation)

hidden units (*e.g.*, in dropout)

weights the loss is not sensitive to small changes in the weight (flat minima)



flat minima generalize better

good performance of SGD using small minibatch is attributed to flat minima

in this case, SGD regularizes the model due to **gradient noise**

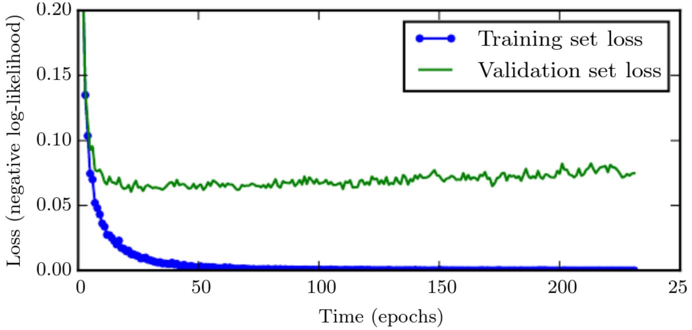
output (avoid overfitting, specially to wrong labels)

a heuristic is to replace hard labels with "soft-labels" label smoothing

e.g., $[0, 0, 1, 0] \rightarrow [\frac{\epsilon}{3}, \frac{\epsilon}{3}, 1-\epsilon, \frac{\epsilon}{3}]$

image credit: Keshkar et al'17

Early stopping

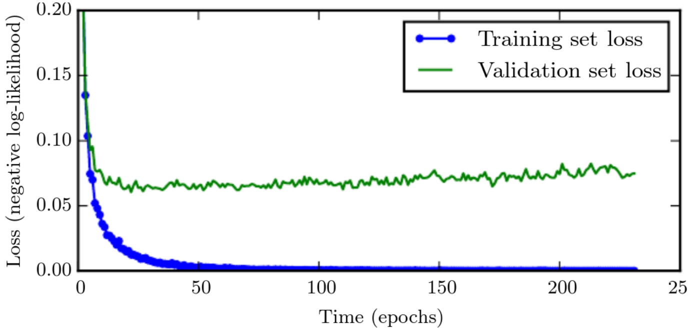


the **test loss**-vs-**time step** is "often" U-shaped

use validation for early stopping

also saves computation!

Early stopping



the **test loss**-vs-**time step** is "often" U-shaped

use validation for early stopping

also saves computation!

early stopping bounds the region of the parameter-space that is reachable in T time-steps

assuming

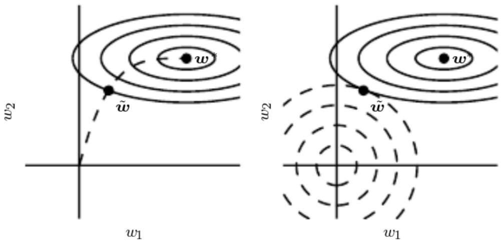
bounded gradient

starting with a small w

it has an effect similar to L2 regularization

we get the regularization path (various λ)

we saw a similar phenomena in boosting



Bagging

several sources of variance in neural networks, such as

- optimization
  - initialization
  - randomness of SGD
  - learning rate and other hyper-parameters
- choice of architecture
  - number of layers, hidden units, etc.

Bagging

several sources of variance in neural networks, such as

- optimization
  - initialization
  - randomness of SGD
  - learning rate and other hyper-parameters
- choice of architecture
  - number of layers, hidden units, etc.

use bagging or even averaging without bootstrap to reduce variance

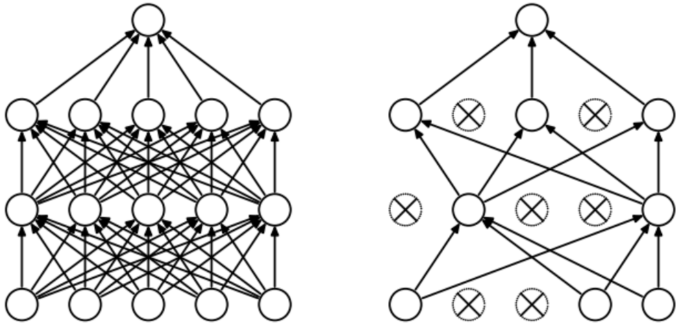
issue: computationally expensive

Dropout

idea

randomly remove a subset of units during training

as opposed to bagging a single model is trained



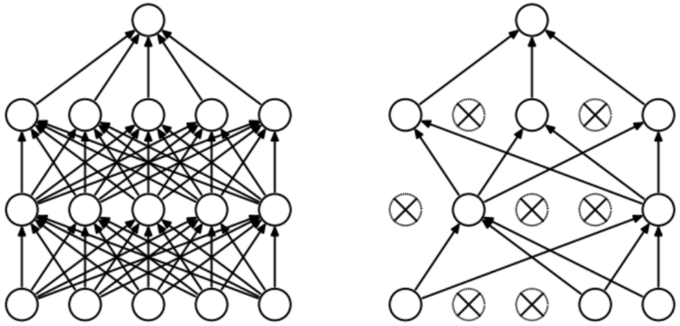
(a) Standard Neural Net (b) After applying dropout.

Dropout

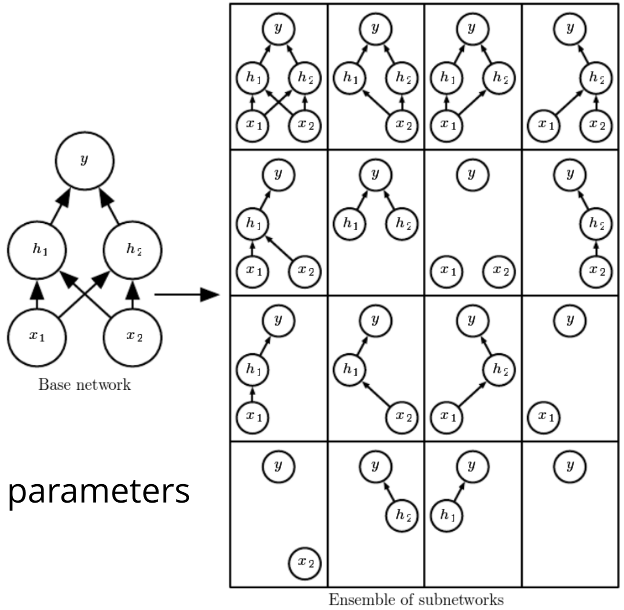
idea

randomly remove a subset of units during training

as opposed to bagging a single model is trained



(a) Standard Neural Net (b) After applying dropout.



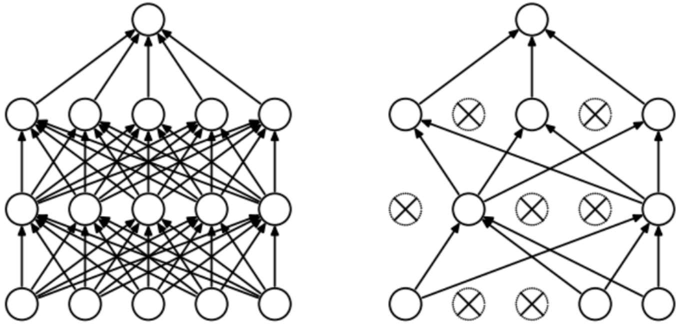
can be viewed as exponentially many subnetworks that share parameters

Dropout

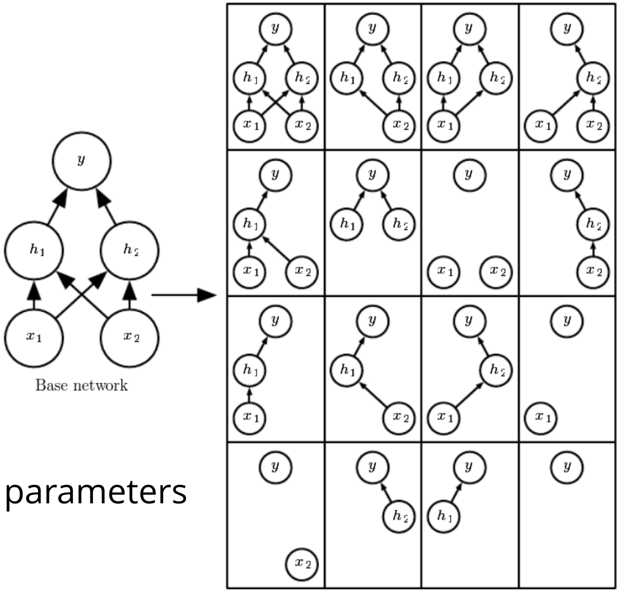
idea

randomly remove a subset of units during training

as opposed to bagging a single model is trained



(a) Standard Neural Net (b) After applying dropout.

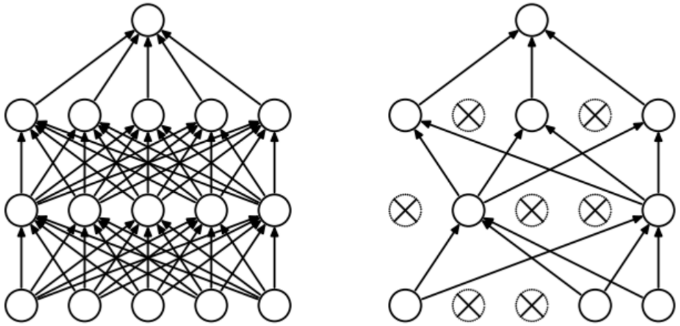


can be viewed as exponentially many subnetworks that share parameters

is one of the most effective regularization schemes for MLPs

Dropout

during training



(a) Standard Neural Net

(b) After applying dropout.

at test time

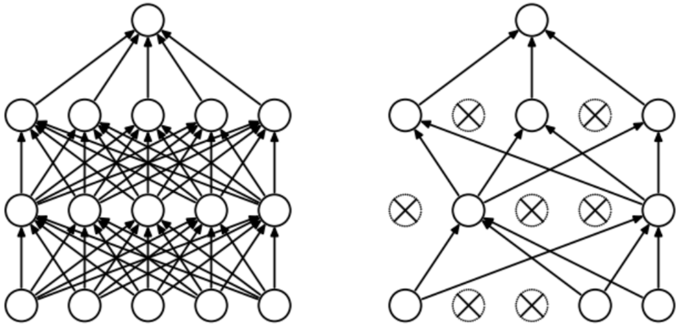
Dropout

during training

for each instance (n):

randomly dropout each unit with probability p (e.g., p=.5)

only the remaining subnetwork participates in training



(a) Standard Neural Net (b) After applying dropout.

at test time

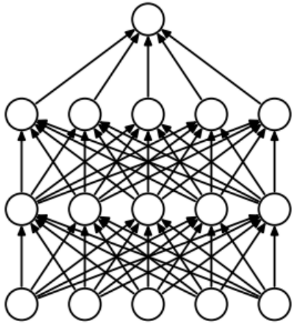
Dropout

during training

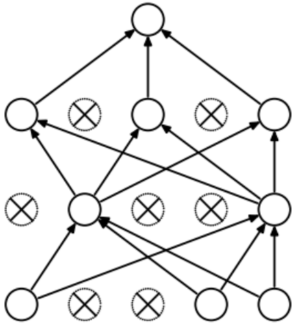
for each instance (n):

randomly dropout each unit with probability p (e.g., p=.5)

only the remaining subnetwork participates in training



(a) Standard Neural Net



(b) After applying dropout.

at test time

ideally we want to average over the prediction of all possible sub-networks

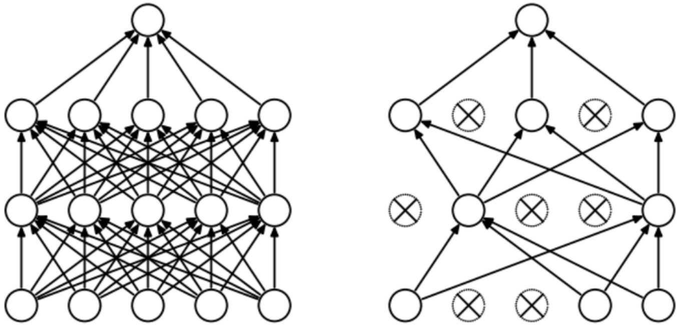
Dropout

during training

for each instance (n):

randomly dropout each unit with probability p (e.g., p=.5)

only the remaining subnetwork participates in training



(a) Standard Neural Net (b) After applying dropout.

at test time

ideally we want to average over the prediction of all possible sub-networks

this is computationally infeasible, instead

1) Monte Carlo dropout: average the prediction of several feed-forward passes using dropout

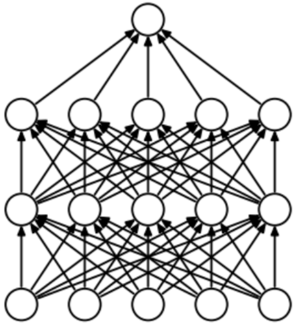
Dropout

during training

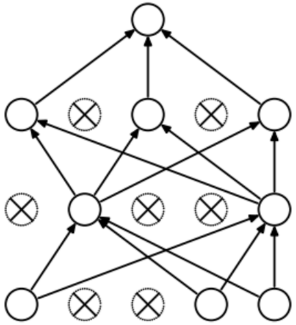
for each instance (n):

randomly dropout each unit with probability p (e.g., p=.5)

only the remaining subnetwork participates in training



(a) Standard Neural Net



(b) After applying dropout.

at test time

ideally we want to average over the prediction of all possible sub-networks

this is computationally infeasible, instead

1) Monte Carlo dropout: average the prediction of several feed-forward passes using dropout

2) weight scaling: scale the weights by **p** to compensate for dropout

e.g., for 50% dropout, scale by a factor of 2

in general this is **not** equivalent to the average prediction of the ensemble

Summary

Deep feed-forward networks learn **adaptive bases**
more complex bases at higher layers
increasing **depth** is often preferable to width
various choices of **activation function** and **architecture**
universal approximation power
their expressive power often necessitates using **regularization** schemes