Statistical Learning– MATH 6333
Set 8 (Neural Networks)

Tamer Oraby

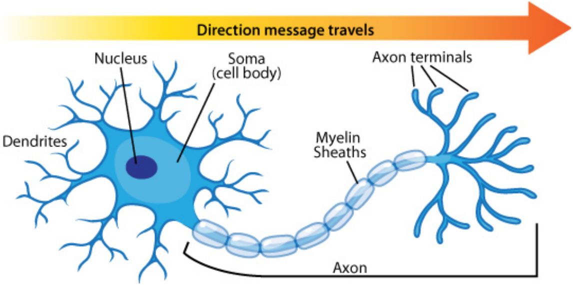
UTRGV

tamer.oraby@utrgv.edu

* Last updated November 23, 2021

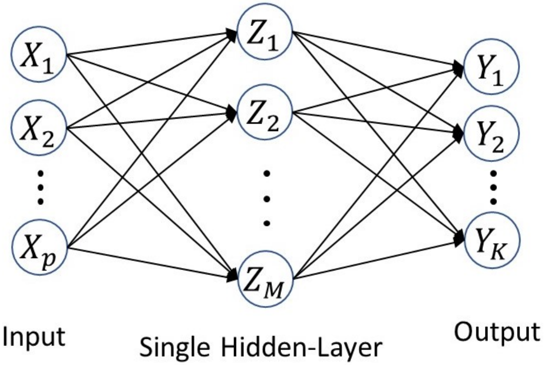
Neural Networks

- Neural Networks started in the 1949 by Hebb.
- Then Rosenblatt introduced artificial neural networks in 1958, with basic units are perceptron that activates or stays inactive upon receiving a signal.
- Back-propagation and training of multi-layer NN in 80s
- Back to be strong in 2000s with better algorithms and devices.



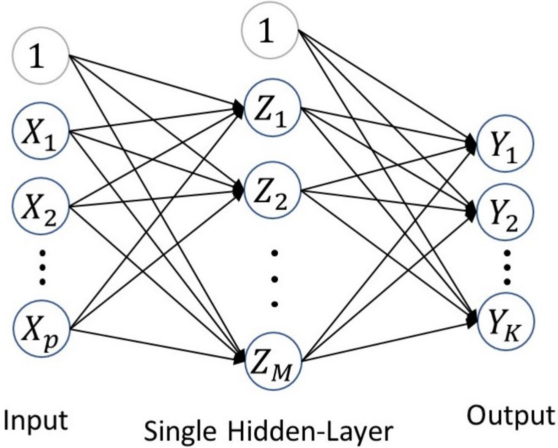
Vanilla Neural Network

- Single hidden layer Neural Network whether for $K-$ response regression or $K-$ class classification

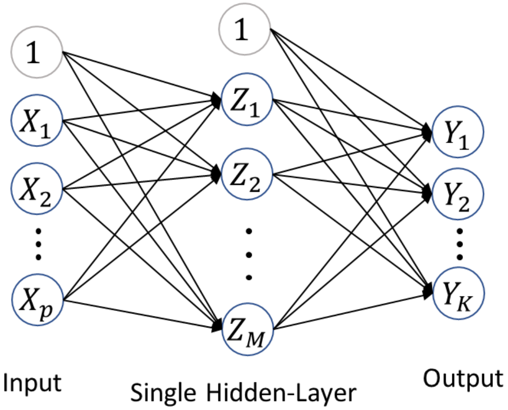


Vanilla Neural Network

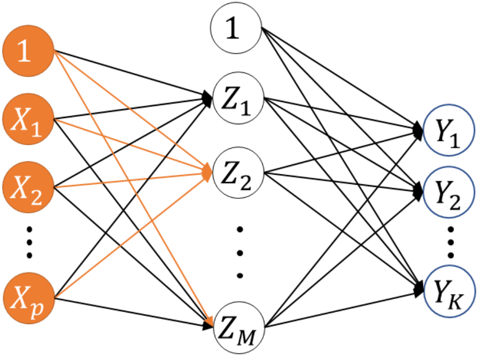
- Single hidden layer Neural Network whether for $K-$ response regression or $K-$ class classification



Vanilla Neural Network

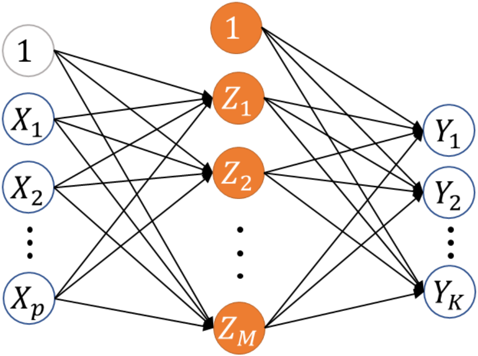


Vanilla Neural Network



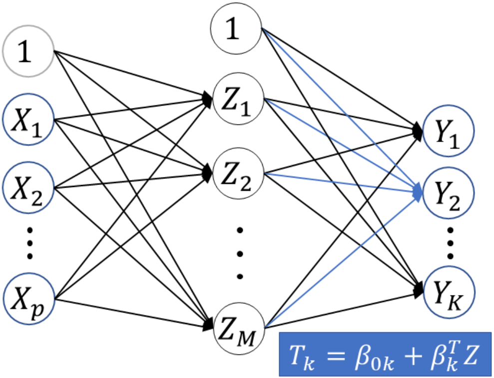
$$\alpha_{0m} + \alpha_m^T X$$

Vanilla Neural Network



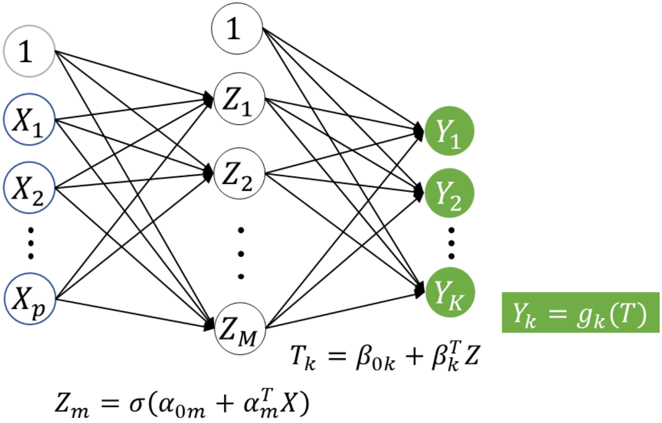
$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X)$$

Vanilla Neural Network



$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X)$$

Vanilla Neural Network



Vanilla Neural Network

That is,

- Input $\mathbf{L} : X \mapsto X_L$ which is linear
- Activation $\sigma : X_L \mapsto Z$
- Target $\mathbf{T} : Z \mapsto T$ which is linear
- Output $g_k : T \mapsto Y$
- In overall, it is a composition of linear (and in a more recent nonlinear) functions

$$Y_k = g_k(\mathbf{T}(\sigma(\mathbf{L}(X)))) =: f_k(x)$$

and $f_k : X \mapsto Y_k$ is a nonlinear transformation of X into Y

Vanilla Neural Network

That is,

- Input $\mathbf{L} : X \mapsto X_L$ which is linear
- Activation $\sigma : X_L \mapsto Z$
- Target $\mathbf{T} : Z \mapsto T$ which is linear
- Output $g_k : T \mapsto Y$
- In overall, it is a composition of linear (and in a more recent nonlinear) functions

$$Y_k = g_k(\mathbf{T}(\sigma(\mathbf{L}(X)))) =: f_k(x)$$

and $f_k : X \mapsto Y_k$ is a nonlinear transformation of X into Y

Vanilla Neural Network

That is,

- Input $\mathbf{L} : X \mapsto X_L$ which is linear
- Activation $\sigma : X_L \mapsto Z$
- Target $\mathbf{T} : Z \mapsto T$ which is linear
- Output $g_k : T \mapsto Y$
- In overall, it is a composition of linear (and in a more recent nonlinear) functions

$$Y_k = g_k(\mathbf{T}(\sigma(\mathbf{L}(X)))) =: f_k(x)$$

and $f_k : X \mapsto Y_k$ is a nonlinear transformation of X into Y

Vanilla Neural Network

That is,

- Input $\mathbf{L} : X \mapsto X_L$ which is linear
- Activation $\sigma : X_L \mapsto Z$
- Target $\mathbf{T} : Z \mapsto T$ which is linear
- Output $g_k : T \mapsto Y$
- In overall, it is a composition of linear (and in a more recent nonlinear) functions

$$Y_k = g_k(\mathbf{T}(\sigma(\mathbf{L}(X)))) =: f_k(x)$$

and $f_k : X \mapsto Y_k$ is a nonlinear transformation of X into Y

Vanilla Neural Network

That is,

- Input $\mathbf{L} : X \mapsto X_L$ which is linear
- Activation $\sigma : X_L \mapsto Z$
- Target $\mathbf{T} : Z \mapsto T$ which is linear
- Output $g_k : T \mapsto Y$
- In overall, it is a composition of linear (and in a more recent nonlinear) functions

$$Y_k = g_k(\mathbf{T}(\sigma(\mathbf{L}(X)))) =: f_k(x)$$

and $f_k : X \mapsto Y_k$ is a nonlinear transformation of X into Y

Vanilla Neural Network

- The activation function σ could be
  - Identity: $\sigma(x) = x$
  - Sigmoid: $\sigma(x) = S(x) = \frac{1}{1+e^{-x}}$
  - Hyperbolic tangent: $\sigma(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - Rectified Linear Unit: $\sigma(x) = ReLU(x) = max(x, 0) = x_+$
  - Rectified softplus: $\sigma(x) = ReSP(x) = \log(1 + e^x)$
- The output function g_k could be
  - k^{th} element "identity" function: $g_k(T) = T_k$
  - Softmax function: $g_k(T) = \frac{e^{T_k}}{\sum_{\ell=1}^{K} e^{T_\ell}}$

Vanilla Neural Network

- The activation function σ could be
  - Identity: $\sigma(x) = x$
  - Sigmoid: $\sigma(x) = S(x) = \frac{1}{1+e^{-x}}$
  - Hyperbolic tangent: $\sigma(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - Rectified Linear Unit: $\sigma(x) = ReLU(x) = max(x, 0) = x_+$
  - Rectified softplus: $\sigma(x) = ReSP(x) = \log(1 + e^x)$
- The output function g_k could be
  - k^{th} element "identity" function: $g_k(T) = T_k$
  - Softmax function: $g_k(T) = \frac{e^{T_k}}{\sum_{\ell=1}^{K} e^{T_\ell}}$

Vanilla Neural Network

- The activation function σ could be
  - Identity: $\sigma(x) = x$
  - Sigmoid: $\sigma(x) = S(x) = \frac{1}{1+e^{-x}}$
  - Hyperbolic tangent: $\sigma(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - Rectified Linear Unit: $\sigma(x) = ReLU(x) = max(x, 0) = x_+$
  - Rectified softplus: $\sigma(x) = ReSP(x) = \log(1 + e^x)$
- The output function g_k could be
  - k^{th} element "identity" function: $g_k(T) = T_k$
  - Softmax function: $g_k(T) = \frac{e^{T_k}}{\sum_{\ell=1}^{K} e^{T_\ell}}$

Vanilla Neural Network

- The activation function σ could be
  - Identity: $\sigma(x) = x$
  - Sigmoid: $\sigma(x) = S(x) = \frac{1}{1+e^{-x}}$
  - Hyperbolic tangent: $\sigma(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - Rectified Linear Unit: $\sigma(x) = ReLU(x) = max(x, 0) = x_+$
  - Rectified softplus: $\sigma(x) = ReSP(x) = \log(1 + e^x)$
- The output function g_k could be
  - k^{th} element "identity" function: $g_k(T) = T_k$
  - Softmax function: $g_k(T) = \frac{e^{T_k}}{\sum_{\ell=1}^{K} e^{T_\ell}}$

Vanilla Neural Network

- The activation function σ could be
  - Identity: $\sigma(x) = x$
  - Sigmoid: $\sigma(x) = S(x) = \frac{1}{1+e^{-x}}$
  - Hyperbolic tangent: $\sigma(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - Rectified Linear Unit: $\sigma(x) = ReLU(x) = max(x, 0) = x_+$
  - Rectified softplus: $\sigma(x) = ReSP(x) = \log(1 + e^x)$
- The output function g_k could be
  - k^{th} element "identity" function: $g_k(T) = T_k$
  - Softmax function: $g_k(T) = \frac{e^{T_k}}{\sum_{\ell=1}^{K} e^{T_\ell}}$

Vanilla Neural Network

- The activation function σ could be
  - Identity: $\sigma(x) = x$
  - Sigmoid: $\sigma(x) = S(x) = \frac{1}{1+e^{-x}}$
  - Hyperbolic tangent: $\sigma(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - Rectified Linear Unit: $\sigma(x) = ReLU(x) = max(x, 0) = x_+$
  - Rectified softplus: $\sigma(x) = ReSP(x) = \log(1 + e^x)$
- The output function g_k could be
  - k^{th} element "identity" function: $g_k(T) = T_k$
  - Softmax function: $g_k(T) = \frac{e^{T_k}}{\sum_{\ell=1}^{K} e^{T_\ell}}$

Vanilla Neural Network

- The activation function σ could be
  - Identity: $\sigma(x) = x$
  - Sigmoid: $\sigma(x) = S(x) = \frac{1}{1+e^{-x}}$
  - Hyperbolic tangent: $\sigma(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - Rectified Linear Unit: $\sigma(x) = ReLU(x) = max(x, 0) = x_+$
  - Rectified softplus: $\sigma(x) = ReSP(x) = \log(1 + e^x)$
- The output function g_k could be
  - k^{th} element "identity" function: $g_k(T) = T_k$
  - Softmax function: $g_k(T) = \frac{e^{T_k}}{\sum_{\ell=1}^{K} e^{T_\ell}}$

Vanilla Neural Network

- The activation function σ could be
  - Identity: $\sigma(x) = x$
  - Sigmoid: $\sigma(x) = S(x) = \frac{1}{1+e^{-x}}$
  - Hyperbolic tangent: $\sigma(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - Rectified Linear Unit: $\sigma(x) = ReLU(x) = max(x, 0) = x_+$
  - Rectified softplus: $\sigma(x) = ReSP(x) = \log(1 + e^x)$
- The output function g_k could be
  - k^{th} element "identity" function: $g_k(T) = T_k$
  - Softmax function: $g_k(T) = \frac{e^{T_k}}{\sum_{\ell=1}^{K} e^{T_\ell}}$

Vanilla Neural Network

- The activation function σ could be
  - Identity: $\sigma(x) = x$
  - Sigmoid: $\sigma(x) = S(x) = \frac{1}{1+e^{-x}}$
  - Hyperbolic tangent: $\sigma(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - Rectified Linear Unit: $\sigma(x) = ReLU(x) = max(x, 0) = x_+$
  - Rectified softplus: $\sigma(x) = ReSP(x) = \log(1 + e^x)$
- The output function g_k could be
  - k^{th} element "identity" function: $g_k(T) = T_k$
  - Softmax function: $g_k(T) = \frac{e^{T_k}}{\sum_{\ell=1}^{K} e^{T_\ell}}$

Fitting Neural Network – Back-propagation

Back-propagation

aka Delta rule

How will we find the $M(p+1)+K(M+1)$ weights

$$\{\alpha_{0m}, \alpha_{jm}, \beta_{0k}, \beta_{mk} : j = 1, \ldots, p; m = 1, \ldots, M; k = 1, \ldots, K\}$$

- Regression problem: minimize Sum-of-squared errors

$$R(\theta) = \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - f_k(x_i))^2 =: \sum_{i=1}^{N} R_i$$

- Classification problem: minimize Sum-of-squared errors (above) or the cross-entropy (deviance)

$$R(\theta) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log(f_k(x_i))$$

with $G(x) = argmax_k f_k(x)$

Back-propagation

aka Delta rule

How will we find the $M(p+1)+K(M+1)$ weights

$$\{\alpha_{0m}, \alpha_{jm}, \beta_{0k}, \beta_{mk} : j = 1, \ldots, p; m = 1, \ldots, M; k = 1, \ldots, K\}$$

- Regression problem: minimize Sum-of-squared errors

$$R(\theta) = \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - f_k(x_i))^2 =: \sum_{i=1}^{N} R_i$$

- Classification problem: minimize Sum-of-squared errors (above) or the cross-entropy (deviance)

$$R(\theta) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log(f_k(x_i))$$

with $G(x) = argmax_k f_k(x)$

Back-propagation

- A gradient descent method called back-propagation is used. (Chain rule from calculus is required.)
- The gradient is
  - $$\frac{\partial R_i}{\partial \beta_{km}} = -2(y_{ik} - f_k(x_i))g_k'(\beta_{0k} + \beta_k^T z_i)z_{mi} = \delta_{ki} z_{mi}$$
    with $z_{0i} = 1$ and δ_{ki} is called the output error
  - and
    $$\begin{aligned}\frac{\partial R_i}{\partial \alpha_{m\ell}} &= -2\sum_{k=1}^{K}(y_{ik} - f_k(x_i))g_k'(\beta_{0k} + \beta_k^T z_i)\beta_{km}\sigma'(\alpha_{0m} + \alpha_m^T x_i)x_{i\ell} \\ &= s_{mi} x_{i\ell}\end{aligned}$$
    with $x_{i0} = 1$ and s_{mi} is called the hidden layer error
  - The back-propagation equations:
    $$s_{mi} = \sigma'(\alpha_{0m} + \alpha_m^T x_i)\sum_{k=1}^{K}\beta_{km}\delta_{ki}$$

Back-propagation

- A gradient descent method called back-propagation is used. (Chain rule from calculus is required.)
- The gradient is
  - $$\frac{\partial R_i}{\partial \beta_{km}} = -2(y_{ik} - f_k(x_i))g_k'(\beta_{0k} + \beta_k^T z_i)z_{mi} = \delta_{ki} z_{mi}$$
    with $z_{0i} = 1$ and δ_{ki} is called the output error
  - and
    $$\frac{\partial R_i}{\partial \alpha_{m\ell}} = -2\sum_{k=1}^{K}(y_{ik} - f_k(x_i))g_k'(\beta_{0k} + \beta_k^T z_i)\beta_{km}\sigma'(\alpha_{0m} + \alpha_m^T x_i)x_{i\ell}$$
    $$= s_{mi} x_{i\ell}$$
    with $x_{i0} = 1$ and s_{mi} is called the hidden layer error
  - The back-propagation equations:
    $$s_{mi} = \sigma'(\alpha_{0m} + \alpha_m^T x_i)\sum_{k=1}^{K}\beta_{km}\delta_{ki}$$

Back-propagation

- A gradient descent method called back-propagation is used. (Chain rule from calculus is required.)
- The gradient is
  - $$\frac{\partial R_i}{\partial \beta_{km}} = -2(y_{ik} - f_k(x_i))g_k'(\beta_{0k} + \beta_k^T z_i)z_{mi} = \delta_{ki} z_{mi}$$
    with $z_{0i} = 1$ and δ_{ki} is called the output error
  - and
    $$\frac{\partial R_i}{\partial \alpha_{m\ell}} = -2\sum_{k=1}^{K}(y_{ik} - f_k(x_i))g_k'(\beta_{0k} + \beta_k^T z_i)\beta_{km}\sigma'(\alpha_{0m} + \alpha_m^T x_i)x_{i\ell}$$
    $$= s_{mi} x_{i\ell}$$
    with $x_{i0} = 1$ and s_{mi} is called the hidden layer error
  - The back-propagation equations:
    $$s_{mi} = \sigma'(\alpha_{0m} + \alpha_m^T x_i)\sum_{k=1}^{K}\beta_{km}\delta_{ki}$$

Back-propagation

- A gradient descent method called back-propagation is used. (Chain rule from calculus is required.)
- The gradient is
  - $$\frac{\partial R_i}{\partial \beta_{km}} = -2(y_{ik} - f_k(x_i))g_k'(\beta_{0k} + \beta_k^T z_i)z_{mi} = \delta_{ki} z_{mi}$$
    with $z_{0i} = 1$ and δ_{ki} is called the output error
  - and
    $$\begin{aligned}\frac{\partial R_i}{\partial \alpha_{m\ell}} &= -2\sum_{k=1}^{K}(y_{ik} - f_k(x_i))g_k'(\beta_{0k} + \beta_k^T z_i)\beta_{km}\sigma'(\alpha_{0m} + \alpha_m^T x_i)x_{i\ell} \\ &= s_{mi} x_{i\ell}\end{aligned}$$
    with $x_{i0} = 1$ and s_{mi} is called the hidden layer error
  - The back-propagation equations:
    $$s_{mi} = \sigma'(\alpha_{0m} + \alpha_m^T x_i)\sum_{k=1}^{K}\beta_{km}\delta_{ki}$$

Back-propagation

- A gradient descent method called back-propagation is used. (Chain rule from calculus is required.)
- The gradient is
  - $$\frac{\partial R_i}{\partial \beta_{km}} = -2(y_{ik} - f_k(x_i))g_k'(\beta_{0k} + \beta_k^T z_i)z_{mi} = \delta_{ki} z_{mi}$$
    with $z_{0i} = 1$ and δ_{ki} is called the output error
  - and
    $$\begin{aligned}\frac{\partial R_i}{\partial \alpha_{m\ell}} &= -2\sum_{k=1}^{K}(y_{ik} - f_k(x_i))g_k'(\beta_{0k} + \beta_k^T z_i)\beta_{km}\sigma'(\alpha_{0m} + \alpha_m^T x_i)x_{i\ell} \\ &= s_{mi} x_{i\ell}\end{aligned}$$
    with $x_{i0} = 1$ and s_{mi} is called the hidden layer error
  - The back-propagation equations:
    $$s_{mi} = \sigma'(\alpha_{0m} + \alpha_m^T x_i)\sum_{k=1}^{K}\beta_{km}\delta_{ki}$$

Back-propagation

- The gradient descent (r+1) update is

$$\beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \beta_{km}^{(r)}}$$

$$= \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \delta_{ki}^{(r)} z_{mi}$$

and

$$\alpha_{m\ell}^{(r+1)} = \alpha_{m\ell}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \alpha_{m\ell}^{(r)}}$$

$$= \alpha_{m\ell}^{(r)} - \gamma_r \sum_{i=1}^{N} s_{mi}^{(r)} x_{i\ell}$$

where γ_r is the learning rate. Updates are batch learning.

Back-propagation

- To perform the the r+1 updated, gradients are updated using a two-pass algorithm

Forward Pass: Inputs are feed into the NN and let them propagate forward to the output and calculate $f_k^{(r)}(x_i)$ based on which

$$\delta_{ki}^{(r)} = -2(y_{ik} - f_k^{(r)}(x_i))g_k'(\beta_{0k}^{(r)} + \beta_k^{(r)T} z_i)$$

are calculated

Backward Pass: Then propagated backward using the back-propagation equations

$$s_{mi}^{(r)} = \sigma'(\alpha_{0m}^{(r)} + \alpha_m^{(r)T} x_i) \sum_{k=1}^{K} \beta_{km}^{(r)} \delta_{ki}^{(r)}$$

Back-propagation

- To perform the the r+1 updated, gradients are updated using a two-pass algorithm

Forward Pass Inputs are feed into the NN and let them propagate forward to the output and calculate $f_k^{(r)}(x_i)$ based on which

$$\delta_{ki}^{(r)} = -2(y_{ik} - f_k^{(r)}(x_i))g_k'(\beta_{0k}^{(r)} + \beta_k^{(r)T} z_i)$$

are calculated

Backward Pass Then propagated backward using the back-propagation equations

$$s_{mi}^{(r)} = \sigma'(\alpha_{0m}^{(r)} + \alpha_m^{(r)T} x_i) \sum_{k=1}^{K} \beta_{km}^{(r)} \delta_{ki}^{(r)}$$

Back-propagation

- To perform the the r+1 updated, gradients are updated using a two-pass algorithm

Forward Pass Inputs are feed into the NN and let them propagate forward to the output and calculate $f_k^{(r)}(x_i)$ based on which

$$\delta_{ki}^{(r)} = -2(y_{ik} - f_k^{(r)}(x_i))g_k'(\beta_{0k}^{(r)} + \beta_k^{(r)\,T} z_i)$$

are calculated

Backward Pass Then propagated backward using the back-propagation equations

$$s_{mi}^{(r)} = \sigma'(\alpha_{0m}^{(r)} + \alpha_m^{(r)\,T} x_i) \sum_{k=1}^{K} \beta_{km}^{(r)} \delta_{ki}^{(r)}$$

Back-propagation

Some notes:

- Training can happen online – one case at a time and update the gradients after each observation and cycling through them many times.
- A training epoch is one sweep through the entire training set.
- It is good to handle big data and data as they arrive to the NN
- The learning rates are γ_r are constant or found using a line search that minimizes the error function at each update. Then it will decrease to zero as r goes to infinity.
- Usually starting points are randomly selected near zero (almost linear functionals) but never zero or NN will not move.
- Stopping rules are needed to avoid overfitting.

Back-propagation

Some notes:

- Training can happen online – one case at a time and update the gradients after each observation and cycling through them many times.
- A training epoch is one sweep through the entire training set.
- It is good to handle big data and data as they arrive to the NN
- The learning rates are γ_r are constant or found using a line search that minimizes the error function at each update. Then it will decrease to zero as r goes to infinity.
- Usually starting points are randomly selected near zero (almost linear functionals) but never zero or NN will not move.
- Stopping rules are needed to avoid overfitting.

Back-propagation

Some notes:

- Training can happen online – one case at a time and update the gradients after each observation and cycling through them many times.
- A training epoch is one sweep through the entire training set.
- It is good to handle big data and data as they arrive to the NN
- The learning rates are γ_r are constant or found using a line search that minimizes the error function at each update. Then it will decrease to zero as r goes to infinity.
- Usually starting points are randomly selected near zero (almost linear functionals) but never zero or NN will not move.
- Stopping rules are needed to avoid overfitting.

Back-propagation

Some notes:

- Training can happen online – one case at a time and update the gradients after each observation and cycling through them many times.
- A training epoch is one sweep through the entire training set.
- It is good to handle big data and data as they arrive to the NN
- The learning rates are γ_r are constant or found using a line search that minimizes the error function at each update. Then it will decrease to zero as r goes to infinity.
- Usually starting points are randomly selected near zero (almost linear functionals) but never zero or NN will not move.
- Stopping rules are needed to avoid overfitting.

Back-propagation

Some notes:

- Training can happen online – one case at a time and update the gradients after each observation and cycling through them many times.
- A training epoch is one sweep through the entire training set.
- It is good to handle big data and data as they arrive to the NN
- The learning rates are γ_r are constant or found using a line search that minimizes the error function at each update. Then it will decrease to zero as r goes to infinity.
- Usually starting points are randomly selected near zero (almost linear functionals) but never zero or NN will not move.
- Stopping rules are needed to avoid overfitting.

Back-propagation

Some notes:

- Training can happen online – one case at a time and update the gradients after each observation and cycling through them many times.
- A training epoch is one sweep through the entire training set.
- It is good to handle big data and data as they arrive to the NN
- The learning rates are γ_r are constant or found using a line search that minimizes the error function at each update. Then it will decrease to zero as r goes to infinity.
- Usually starting points are randomly selected near zero (almost linear functionals) but never zero or NN will not move.
- Stopping rules are needed to avoid overfitting.

Back-propagation

Some notes:

- Consider regularization after standardization of the inputs

$$R(\theta) + \lambda J(\theta)$$

- Weight decay (L_2-) penalty:

$$J(\theta) = \sum_{mk} \beta_{mk}^2 + \sum_{jm} \alpha_{jm}^2$$

- Weight elimination penalty:

$$J(\theta) = \sum_{mk} \frac{\beta_{mk}^2}{1 + \beta_{mk}^2} + \sum_{jm} \frac{\alpha_{jm}^2}{1 + \alpha_{jm}^2}$$

more shrinkage than weight decay penalty

Back-propagation

Some notes:

- Consider regularization after standardization of the inputs

$$R(\theta) + \lambda J(\theta)$$

  - Weight decay (L_2-) penalty:

$$J(\theta) = \sum_{mk} \beta_{mk}^2 + \sum_{jm} \alpha_{jm}^2$$

  - Weight elimination penalty:

$$J(\theta) = \sum_{mk} \frac{\beta_{mk}^2}{1 + \beta_{mk}^2} + \sum_{jm} \frac{\alpha_{jm}^2}{1 + \alpha_{jm}^2}$$

    more shrinkage than weight decay penalty

Back-propagation

Some notes:

- Consider regularization after standardization of the inputs

$$R(\theta) + \lambda J(\theta)$$

  - Weight decay (L_2-) penalty:

$$J(\theta) = \sum_{mk} \beta_{mk}^2 + \sum_{jm} \alpha_{jm}^2$$

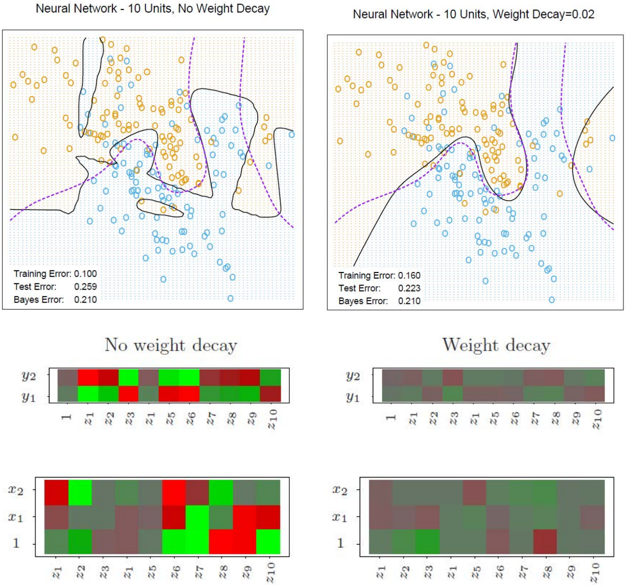
  - Weight elimination penalty:

$$J(\theta) = \sum_{mk} \frac{\beta_{mk}^2}{1+\beta_{mk}^2} + \sum_{jm} \frac{\alpha_{jm}^2}{1+\alpha_{jm}^2}$$

    more shrinkage than weight decay penalty

Back-propagation

Example (Orange vs Blue)



Back-propagation

More notes:

- Again, standardize inputs.
- More nodes and hidden layers is better then less.
- Consider bagging as there might be more than one minima of $R(\theta)$.
- The learning rates are γ_r are constant or found using a line search that minimizes the error function at each update. Then it will decrease to zero as r goes to infinity.
- Usually starting points are randomly selected near zero (almost linear functionals) but never zero or NN will not move.
- Stopping rules are needed to avoid overfitting.

Back-propagation

More notes:

- Again, standardize inputs.
- More nodes and hidden layers is better then less.
- Consider bagging as there might be more than one minima of $R(\theta)$.
- The learning rates are γ_r are constant or found using a line search that minimizes the error function at each update. Then it will decrease to zero as r goes to infinity.
- Usually starting points are randomly selected near zero (almost linear functionals) but never zero or NN will not move.
- Stopping rules are needed to avoid overfitting.

Back-propagation

More notes:

- Again, standardize inputs.
- More nodes and hidden layers is better then less.
- Consider bagging as there might be more than one minima of $R(\theta)$.
- The learning rates are γ_r are constant or found using a line search that minimizes the error function at each update. Then it will decrease to zero as r goes to infinity.
- Usually starting points are randomly selected near zero (almost linear functionals) but never zero or NN will not move.
- Stopping rules are needed to avoid overfitting.

Back-propagation

More notes:

- Again, standardize inputs.
- More nodes and hidden layers is better then less.
- Consider bagging as there might be more than one minima of $R(\theta)$.
- The learning rates are γ_r are constant or found using a line search that minimizes the error function at each update. Then it will decrease to zero as r goes to infinity.
- Usually starting points are randomly selected near zero (almost linear functionals) but never zero or NN will not move.
- Stopping rules are needed to avoid overfitting.

Back-propagation

More notes:

- Again, standardize inputs.
- More nodes and hidden layers is better then less.
- Consider bagging as there might be more than one minima of $R(\theta)$.
- The learning rates are γ_r are constant or found using a line search that minimizes the error function at each update. Then it will decrease to zero as r goes to infinity.
- Usually starting points are randomly selected near zero (almost linear functionals) but never zero or NN will not move.
- Stopping rules are needed to avoid overfitting.

Back-propagation

More notes:

- Again, standardize inputs.
- More nodes and hidden layers is better then less.
- Consider bagging as there might be more than one minima of $R(\theta)$.
- The learning rates are γ_r are constant or found using a line search that minimizes the error function at each update. Then it will decrease to zero as r goes to infinity.
- Usually starting points are randomly selected near zero (almost linear functionals) but never zero or NN will not move.
- Stopping rules are needed to avoid overfitting.

Back -propagation

DIY in R

1. Carry out a neural network analysis of the orange vs blue data using nnet in R.
2. Carry out a neural network analysis of the ZIP Code data

Please study Example 11.7 in the ESL textbook.

Back -propagation

DIY in R

1. Carry out a neural network analysis of the orange vs blue data using nnet in R.
2. Carry out a neural network analysis of the ZIP Code data

Please study Example 11.7 in the ESL textbook.

Back -propagation

DIY in R

1. Carry out a neural network analysis of the orange vs blue data using nnet in R.
2. Carry out a neural network analysis of the ZIP Code data

Please study Example 11.7 in the ESL textbook.

Back -propagation

DIY in R

1. Carry out a neural network analysis of the orange vs blue data using nnet in R.
2. Carry out a neural network analysis of the ZIP Code data

Please study Example 11.7 in the ESL textbook.

Universal Approximation Theorem

Universal Approximation Theorem

Theorem (Cybenko's Universal approximation theorem (1989))

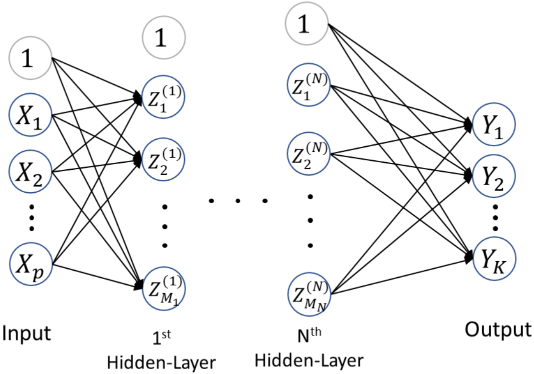
Let f be a $C([0,1]^n)$ (also an L_2- function), and σ be such that $\lim_{x\to\infty}\sigma(x) = 1$ and $\lim_{x\to-\infty}\sigma(x) = 0$, then for some > 0, there is $M = M(\epsilon)$ such that

$$inf_{\alpha,\beta}\left|f(x) - \sum_{m=1}^{M}\beta_m\sigma(\alpha_{0m} + \alpha_m^T x)\right| \leq \epsilon$$

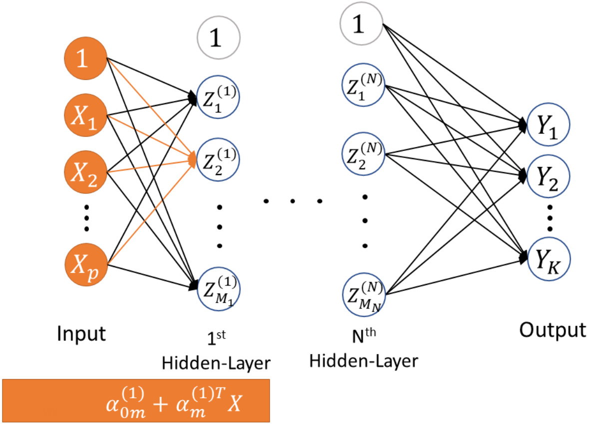
It basically says that any such function f could be approximated by an ANN. The more the hidden-layer's nodes, the better the approximation.

Multi-layer Neural Network

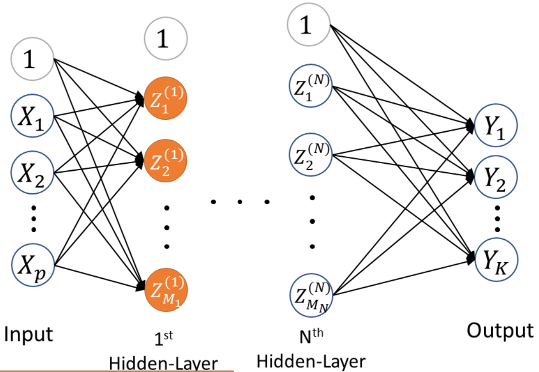
Multi-layer Neural Network



Multi-layer Neural Network

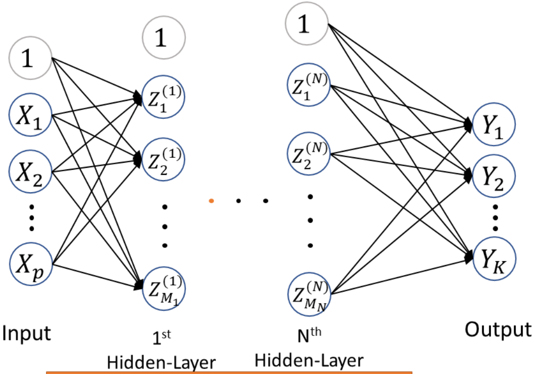


Multi-layer Neural Network



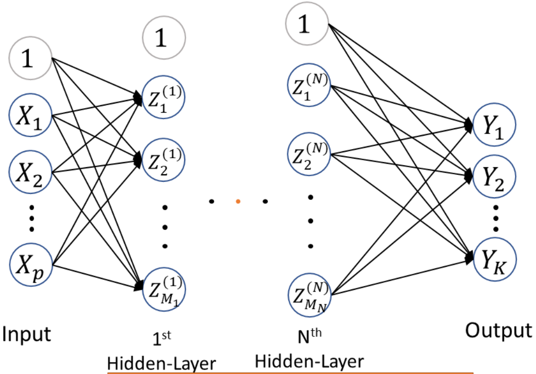
$$Z_m^{(1)} = \sigma^{(1)}\left(\alpha_{0m}^{(1)} + \alpha_m^{(1)T} X\right)$$

Multi-layer Neural Network



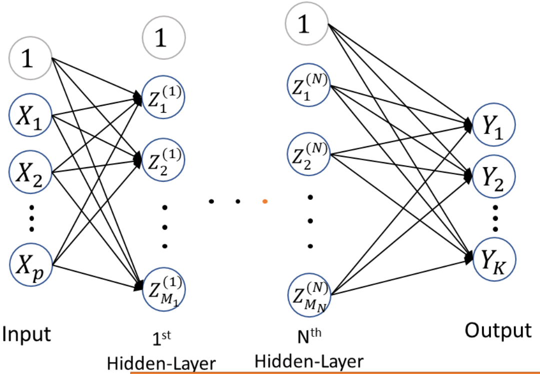
$$Z_m^{(2)} = \sigma^{(2)}\left(\alpha_{0m}^{(2)} + \alpha_m^{(2)T} Z^{(1)}\right)$$

Multi-layer Neural Network



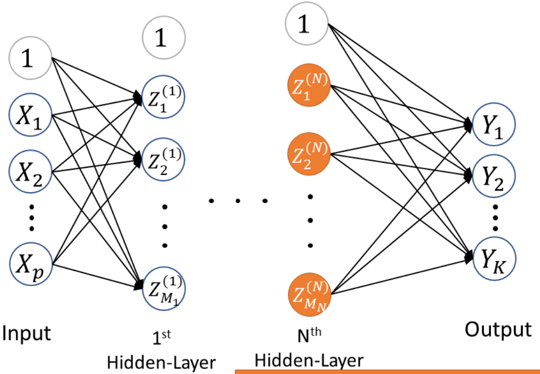
$$Z_m^{(i)} = \sigma^{(i)}\left(\alpha_{0m}^{(i)} + \alpha_m^{(i)T} Z^{(i-1)}\right)$$

Multi-layer Neural Network



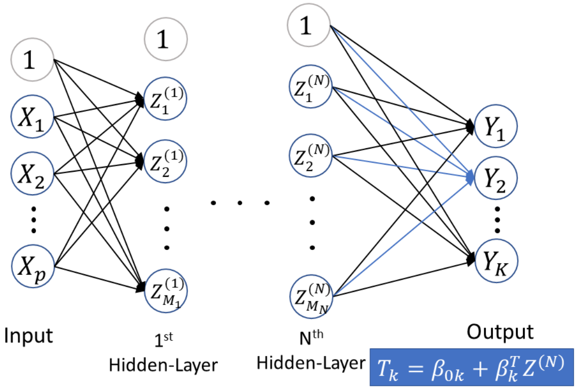
$$Z_m^{(N-1)} = \sigma^{(N-1)}\left(\alpha_{0m}^{(N-1)} + \alpha_m^{(N-1)T} Z^{(N-2)}\right)$$

Multi-layer Neural Network

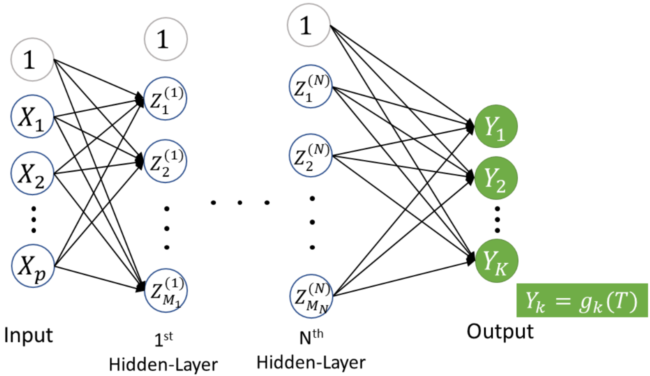


$$Z_m^{(N)} = \sigma^{(N)}\left(\alpha_{0m}^{(N)} + \alpha_m^{(N)T} Z^{(N-1)}\right)$$

Multi-layer Neural Network



Multi-layer Neural Network



Multi-layer Neural Network

That is,

- It is a composition of linear (and in a more recent nonlinear) functions

$$Y = g(\beta\sigma^{(N)}(\alpha^{(N)}\sigma^{(N-1)}(\alpha^{(N-1)}\sigma^{(N-2)}(\cdots\sigma^{(1)}(\alpha^{(1)}x)))))$$
$$=: f(x)$$

where β and $\alpha^{(i)}$ are weight matrices and $f : X \mapsto Y$ is a multi-response nonlinear transformation of X into Y

Multi-layer Neural Network

- Fitting MNN is done using back-propagation.
- By minimizing $R(\theta) = \frac{1}{n}\sum_{i=1}^{n} R_i(\theta)$, which is usually regularized into $R(\theta) + \lambda J(\theta)$. The tuning parameter λ could be determined using CV.
- With gradient descent's iterative formula

$$\theta^{new} = \theta^{old} - \gamma_r \nabla R(\theta)|_{\theta^{old}}$$

- or with stochastic gradient descent's iterative formula

$$\theta^{new} = \theta^{old} - \gamma_r \left(\nabla R(\theta)|_{\theta^{old}} + \epsilon_{new}\right)$$

where ϵ are iid random noise with mean 0.

Multi-layer Neural Network

- Fitting MNN is done using back-propagation.
- By minimizing $R(\theta) = \frac{1}{n}\sum_{i=1}^{n} R_i(\theta)$, which is usually regularized into $R(\theta) + \lambda J(\theta)$. The tuning parameter λ could be determined using CV.
- With gradient descent's iterative formula

$$\theta^{new} = \theta^{old} - \gamma_r \, \nabla R(\theta)|_{\theta^{old}}$$

- or with stochastic gradient descent's iterative formula

$$\theta^{new} = \theta^{old} - \gamma_r \, (\nabla R(\theta)|_{\theta^{old}} + \epsilon_{new})$$

where ϵ are iid random noise with mean 0.

Multi-layer Neural Network

- Fitting MNN is done using back-propagation.
- By minimizing $R(\theta) = \frac{1}{n}\sum_{i=1}^{n} R_i(\theta)$, which is usually regularized into $R(\theta) + \lambda J(\theta)$. The tuning parameter λ could be determined using CV.
- With gradient descent's iterative formula

$$\theta^{new} = \theta^{old} - \gamma_r \nabla R(\theta)|_{\theta^{old}}$$

- or with stochastic gradient descent's iterative formula

$$\theta^{new} = \theta^{old} - \gamma_r \left(\nabla R(\theta)|_{\theta^{old}} + \epsilon_{new}\right)$$

where ϵ are iid random noise with mean 0.

Multi-layer Neural Network

- Fitting MNN is done using back-propagation.
- By minimizing $R(\theta) = \frac{1}{n}\sum_{i=1}^{n} R_i(\theta)$, which is usually regularized into $R(\theta) + \lambda J(\theta)$. The tuning parameter λ could be determined using CV.
- With gradient descent's iterative formula

$$\theta^{new} = \theta^{old} - \gamma_r \nabla R(\theta)|_{\theta^{old}}$$

- or with stochastic gradient descent's iterative formula

$$\theta^{new} = \theta^{old} - \gamma_r \left(\nabla R(\theta)|_{\theta^{old}} + \epsilon_{new}\right)$$

  where ϵ are iid random noise with mean 0.

Multi-layer Neural Network

- Another stochastic gradient descent (SGD) ...
- since $\nabla R(\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla R_i(\theta)$,
- then a randomly selected i (as a random epoch) results in a random $R_i(\theta)$ that can be used to update the weights via

$$\theta^{new} = \theta^{old} - \gamma_r \nabla R_i(\theta)|_{\theta^{old}}$$

- A forward pass is made then followed by a number of N backward passes to update the weights.

Multi-layer Neural Network

- Another stochastic gradient descent (SGD) ...
- since $\nabla R(\theta) = \frac{1}{n}\sum_{i=1}^{n} \nabla R_i(\theta)$,
- then a randomly selected i (as a random epoch) results in a random $R_i(\theta)$ that can be used to update the weights via

$$\theta^{new} = \theta^{old} - \gamma_r \nabla R_i(\theta)|_{\theta^{old}}$$

- A forward pass is made then followed by a number of N backward passes to update the weights.

Multi-layer Neural Network

- Another stochastic gradient descent (SGD) ...
- since $\nabla R(\theta) = \frac{1}{n}\sum_{i=1}^{n} \nabla R_i(\theta)$,
- then a randomly selected i (as a random epoch) results in a random $R_i(\theta)$ that can be used to update the weights via

$$\theta^{new} = \theta^{old} - \gamma_r \nabla R_i(\theta)|_{\theta^{old}}$$

- A forward pass is made then followed by a number of N backward passes to update the weights.

Multi-layer Neural Network

- Another stochastic gradient descent (SGD) ...
- since $\nabla R(\theta) = \frac{1}{n}\sum_{i=1}^{n} \nabla R_i(\theta)$,
- then a randomly selected i (as a random epoch) results in a random $R_i(\theta)$ that can be used to update the weights via

$$\theta^{new} = \theta^{old} - \gamma_r \nabla R_i(\theta)|_{\theta^{old}}$$

- A forward pass is made then followed by a number of N backward passes to update the weights.

Bayesian Neural Network

Bayesian Neural Network

Look at dissertation of Neal (1995)

- Given a training data (X_{tr}, y_{tr}), in a regression or classification problem $P(Y|X, \theta)$ can give the posterior distribution for the regression in which the mean or the classification probability is sought
- Given a prior $P(\theta)$, the posterior distribution is given by

$$P(\theta|X_{tr}, y_{tr}) = \frac{P(y_{tr}|X_{tr}, \theta)\, P(\theta)}{\int_\Theta P(y_{tr}|X_{tr}, \theta')\, P(\theta')\, d\theta'}$$

- A prediction for a new case X_* is done using the predictive posterior

$$P(y_*|X_*, X_{tr}, y_{tr}) = \int_\Theta P(y_*|X_*, \theta)\, P(\theta|X_{tr}, y_{tr})\, d\theta$$

Bayesian Neural Network

Look at dissertation of Neal (1995)

- Given a training data (X_{tr}, y_{tr}), in a regression or classification problem $P(Y|X,\theta)$ can give the posterior distribution for the regression in which the mean or the classification probability is sought
- Given a prior $P(\theta)$, the posterior distribution is given by

$$P(\theta|X_{tr}, y_{tr}) = \frac{P(y_{tr}|X_{tr},\theta)\, P(\theta)}{\int_\Theta P(y_{tr}|X_{tr},\theta')\, P(\theta')\, d\theta'}$$

- A prediction for a new case X_* is done using the predictive posterior

$$P(y_*|X_*, X_{tr}, y_{tr}) = \int_\Theta P(y_*|X_*,\theta)\, P(\theta|X_{tr}, y_{tr})\, d\theta$$

Bayesian Neural Network

Look at dissertation of Neal (1995)

- Given a training data (X_{tr}, y_{tr}), in a regression or classification problem $P(Y|X, \theta)$ can give the posterior distribution for the regression in which the mean or the classification probability is sought
- Given a prior $P(\theta)$, the posterior distribution is given by

$$P(\theta|X_{tr}, y_{tr}) = \frac{P(y_{tr}|X_{tr}, \theta)\, P(\theta)}{\int_{\Theta} P(y_{tr}|X_{tr}, \theta')\, P(\theta')\, d\theta'}$$

- A prediction for a new case X_* is done using the predictive posterior

$$P(y_*|X_*, X_{tr}, y_{tr}) = \int_{\Theta} P(y_*|X_*, \theta)\, P(\theta|X_{tr}, y_{tr})\, d\theta$$

Bayesian Neural Network

- A MCMC algorithm is need to simulate from the predictive posterior

$$P(y_*|X_*, X_{tr}, y_{tr}) = \int_\Theta P(y_*|X_*, \theta)\, P(\theta|X_{tr}, y_{tr})\, d\theta$$

with the following steps

  - Use a converged MCMC algorithm to sample from the chain values $\theta^{(j)}$ for $j = 1, 2, \ldots, J$ that are representing simulated values from $P(\theta|X_{tr}, y_{tr})$
  - Then, simulate $y_*^{(j)}$ from $P(y|X_*, \theta^{(j)})$ for $j = 1, 2, \ldots, J$ and use it to make the empirical estimate of the posterior predictive, or
  - instead, calculate $\frac{1}{J}\sum_{j=1}^{J} P(y_*|X_*, \theta^{(j)})$ for sufficiently large J and for a set of values y_*

Bayesian Neural Network

- A MCMC algorithm is need to simulate from the predictive posterior

$$P(y_*|X_*, X_{tr}, y_{tr}) = \int_\Theta P(y_*|X_*, \theta)\, P(\theta|X_{tr}, y_{tr})\, d\theta$$

with the following steps

  - Use a converged MCMC algorithm to sample from the chain values $\theta^{(j)}$ for $j = 1, 2, \ldots, J$ that are representing simulated values from $P(\theta|X_{tr}, y_{tr})$
  - Then, simulate $y_*^{(j)}$ from $P(y|X_*, \theta^{(j)})$ for $j = 1, 2, \ldots, J$ and use it to make the empirical estimate of the posterior predictive, or
  - instead, calculate $\frac{1}{J}\sum_{j=1}^{J} P(y_*|X_*, \theta^{(j)})$ for sufficiently large J and for a set of values y_*

Bayesian Neural Network

- A MCMC algorithm is need to simulate from the predictive posterior

$$P(y_*|X_*, X_{tr}, y_{tr}) = \int_\Theta P(y_*|X_*, \theta)\, P(\theta|X_{tr}, y_{tr})\, d\theta$$

  with the following steps

  - Use a converged MCMC algorithm to sample from the chain values $\theta^{(j)}$ for $j = 1, 2, \ldots, J$ that are representing simulated values from $P(\theta|X_{tr}, y_{tr})$
  - Then, simulate $y_*^{(j)}$ from $P(y|X_*, \theta^{(j)})$ for $j = 1, 2, \ldots, J$ and use it to make the empirical estimate of the posterior predictive, or
  - instead, calculate $\frac{1}{J}\sum_{j=1}^{J} P(y_*|X_*, \theta^{(j)})$ for sufficiently large J and for a set of values y_*

Bayesian Neural Network

- A MCMC algorithm is need to simulate from the predictive posterior

$$P(y_*|X_*, X_{tr}, y_{tr}) = \int_\Theta P(y_*|X_*, \theta)\, P(\theta|X_{tr}, y_{tr})\, d\theta$$

with the following steps

  - Use a converged MCMC algorithm to sample from the chain values $\theta^{(j)}$ for $j = 1, 2, \ldots, J$ that are representing simulated values from $P(\theta|X_{tr}, y_{tr})$
  - Then, simulate $y_*^{(j)}$ from $P(y|X_*, \theta^{(j)})$ for $j = 1, 2, \ldots, J$ and use it to make the empirical estimate of the posterior predictive, or
  - instead, calculate $\frac{1}{J}\sum_{j=1}^{J} P(y_*|X_*, \theta^{(j)})$ for sufficiently large J and for a set of values y_*

Bayesian Neural Network

Neal and Zhang (2006) suggested using

- a hybrid MCMC algorithm
- performing pre-processing of the inputs like using univariate t-tests
- more importantly, use Automatic Relevance Determination (ARD) that is
  - assign (same for each j) priors for the weights $\alpha_{jm}^{(1)}$ from input j in the input layer to the first hidden layer that have mean 0 and variance σ_j^2 (a hyper-parameter)
  - (A regularization step) assign hyper-prior to the hyper-parameters σ_j^2 that has very small variance in which case a weight decay will be happen

Bayesian Neural Network

Neal and Zhang (2006) suggested using

- a hybrid MCMC algorithm
- performing pre-processing of the inputs like using univariate t-tests
- more importantly, use Automatic Relevance Determination (ARD) that is
  - assign (same for each j) priors for the weights $\alpha_{jm}^{(1)}$ from input j in the input layer to the first hidden layer that have mean 0 and variance σ_j^2 (a hyper-parameter)
  - (A regularization step) assign hyper-prior to the hyper-parameters σ_j^2 that has very small variance in which case a weight decay will be happen

Bayesian Neural Network

Neal and Zhang (2006) suggested using

- a hybrid MCMC algorithm
- performing pre-processing of the inputs like using univariate t-tests
- more importantly, use Automatic Relevance Determination (ARD) that is
  - assign (same for each j) priors for the weights $\alpha_{jm}^{(1)}$ from input j in the input layer to the first hidden layer that have mean 0 and variance σ_j^2 (a hyper-parameter)
  - (A regularization step) assign hyper-prior to the hyper-parameters σ_j^2 that has very small variance in which case a weight decay will be happen

Bayesian Neural Network

Neal and Zhang (2006) suggested using

- a hybrid MCMC algorithm
- performing pre-processing of the inputs like using univariate t-tests
- more importantly, use Automatic Relevance Determination (ARD) that is
  - assign (same for each j) priors for the weights $\alpha_{jm}^{(1)}$ from input j in the input layer to the first hidden layer that have mean 0 and variance σ_j^2 (a hyper-parameter)
  - (A regularization step) assign hyper-prior to the hyper-parameters σ_j^2 that has very small variance in which case a weight decay will be happen

Bayesian Neural Network

Neal and Zhang (2006) suggested using

- a hybrid MCMC algorithm
- performing pre-processing of the inputs like using univariate t-tests
- more importantly, use Automatic Relevance Determination (ARD) that is
  - assign (same for each j) priors for the weights $\alpha_{jm}^{(1)}$ from input j in the input layer to the first hidden layer that have mean 0 and variance σ_j^2 (a hyper-parameter)
  - (A regularization step) assign hyper-prior to the hyper-parameters σ_j^2 that has very small variance in which case a weight decay will be happen

Bayesian Neural Network

Example (Neal (1995))

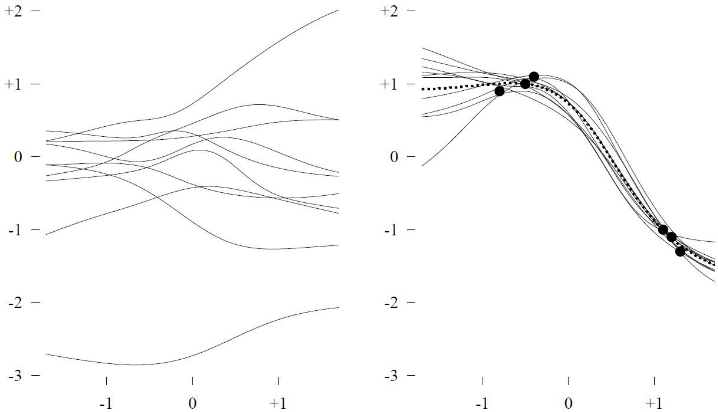
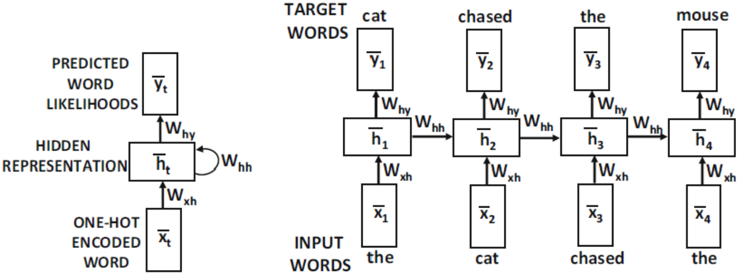


Figure 1.2: An illustration of Bayesian inference for a neural network. On the left are the functions computed by ten networks whose weights and biases were drawn at random from Gaussian prior distributions. On the right are six data points and the functions computed by ten networks drawn from the posterior distribution derived from the prior and the likelihood due to these data points. The heavy dotted line is the average of the ten functions drawn from the posterior, which is an approximation to the function that should be guessed in order to minimize squared error loss.

Recurrent Neural Network

Recurrent Neural Network

Example



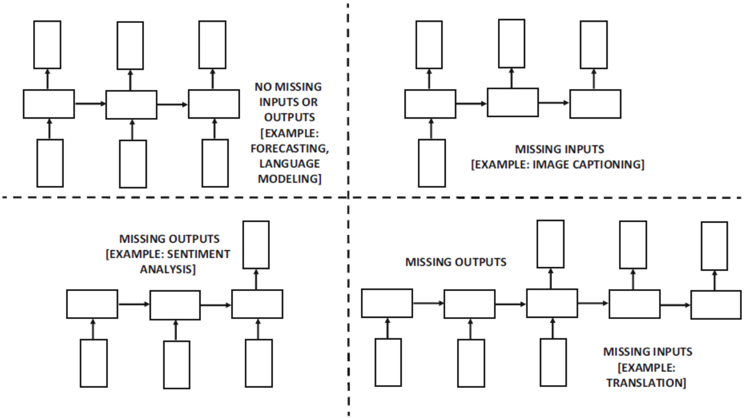
(a) RNN (b) Time-layered representation of (a)

$$\bar{h}_t = tanh(W_{xh}\bar{x}_t + W_{hh}\bar{h}_{t-1})$$

and

$$\bar{y}_t = W_{hy}\bar{h}_t$$

Recurrent Neural Network



Convolutional Neural Network

Convolutional Neural Network

- Data is given in 1, 2, ... dimensional grid, like time series, pictures, ...
- CNN can provide parameter efficient, invariant to transformation, feature extraction learning method.
- There are different famous CNNs: *AlexNet, ZFNet, LeNet-5, GoogLeNet, ResNet*

Convolutional Neural Network

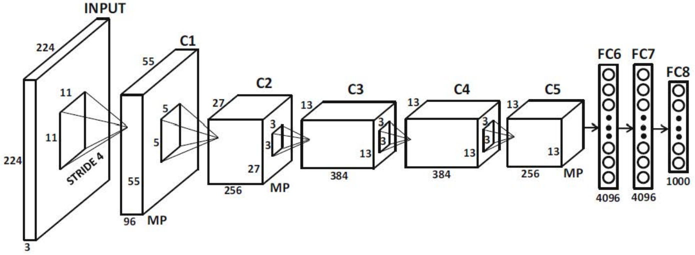
- Data is given in 1, 2, ... dimensional grid, like time series, pictures, ...
- CNN can provide parameter efficient, invariant to transformation, feature extraction learning method.
- There are different famous CNNs: *AlexNet, ZFNet, LeNet-5, GoogLeNet, ResNet*

Convolutional Neural Network

- Data is given in 1, 2, ... dimensional grid, like time series, pictures, ...
- CNN can provide parameter efficient, invariant to transformation, feature extraction learning method.
- There are different famous CNNs: *AlexNet, ZFNet, LeNet-5, GoogLeNet, ResNet*

Convolutional Neural Network

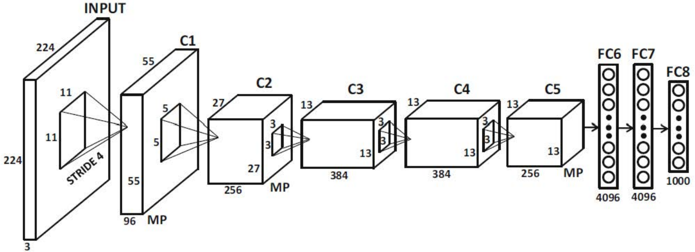
AlexNet



- Input: 224x224x3 images - (3=RGB)
- AlexNet uses 96 filters (of semantic features) of kernel size 11x11x3 in the first layer with stride 4, resulting in a first layer of 55x55x96
- then a max-pool (MP) is applied with 256 filters of kernel size 5x5x96 with stride 2. All MPs are 3x3 stride 2.

Convolutional Neural Network

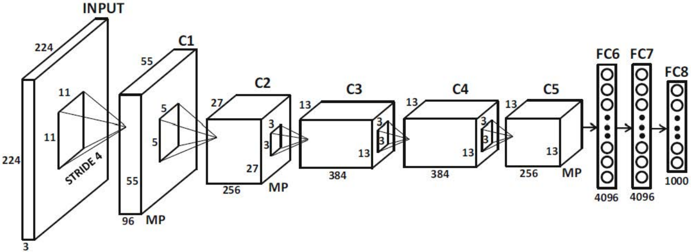
AlexNet



- Input: 224x224x3 images - (3=RGB)
- AlexNet uses 96 filters (of semantic features) of kernel size 11x11x3 in the first layer with stride 4, resulting in a first layer of 55x55x96
- then a max-pool (MP) is applied with 256 filters of kernel size 5x5x96 with stride 2. All MPs are 3x3 stride 2.

Convolutional Neural Network

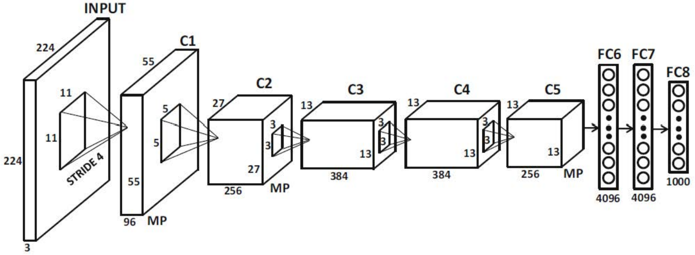
AlexNet



- Input: 224x224x3 images - (3=RGB)
- AlexNet uses 96 filters (of semantic features) of kernel size 11x11x3 in the first layer with stride 4, resulting in a first layer of 55x55x96
- then a max-pool (MP) is applied with 256 filters of kernel size 5x5x96 with stride 2. All MPs are 3x3 stride 2.

Convolutional Neural Network

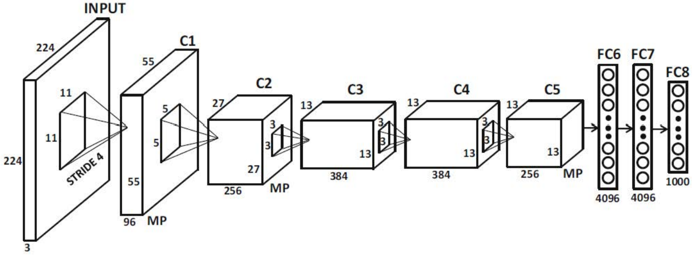
AlexNet



- The rest are 384 filters of kernel size 3x3x256 then 3x3x384, then 256 filters of kernel size 3x3x384
- ReLU activation function and response-normalization are used after every convolution layer and the output uses softmax function for classification.
- *FC7* is a 4096 dimensional representation of the image and can be used as feature extraction from the image (referred to *FC7* features).

Convolutional Neural Network

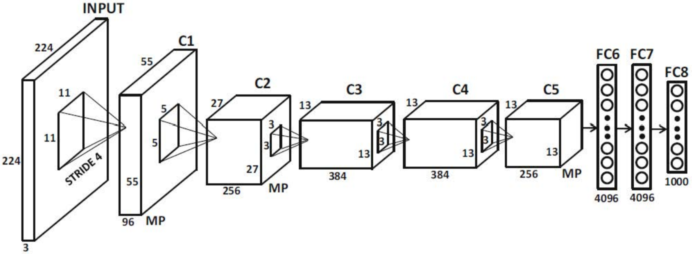
AlexNet



- The rest are 384 filters of kernel size 3x3x256 then 3x3x384, then 256 filters of kernel size 3x3x384
- ReLU activation function and response-normalization are used after every convolution layer and the output uses softmax function for classification.
- *FC7* is a 4096 dimensional representation of the image and can be used as feature extraction from the image (referred to *FC7* features).

Convolutional Neural Network

AlexNet



- The rest are 384 filters of kernel size 3x3x256 then 3x3x384, then 256 filters of kernel size 3x3x384
- ReLU activation function and response-normalization are used after every convolution layer and the output uses softmax function for classification.
- *FC7* is a 4096 dimensional representation of the image and can be used as feature extraction from the image (referred to *FC7* features).

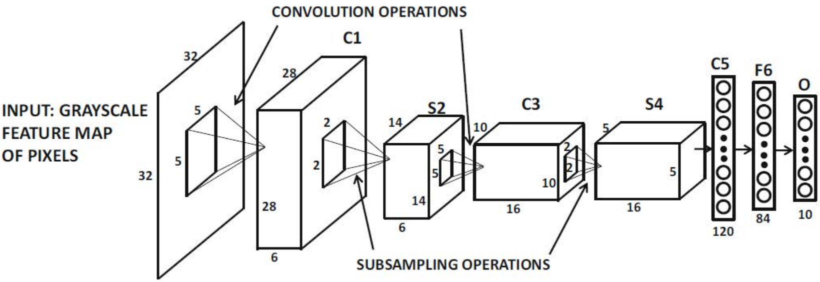
Convolutional Neural Network

ZFNet

	AlexNet	*ZFNet*
Volume: Operations:	$224 \times 224 \times 3$ Conv 11×11 (stride 4)	$224 \times 224 \times 3$ Conv 7×7 (stride 2), MP
Volume: Operations:	$55 \times 55 \times 96$ Conv 5×5, MP	$55 \times 55 \times 96$ Conv 5×5 (stride 2), MP
Volume: Operations:	$27 \times 27 \times 256$ Conv 3×3, MP	$13 \times 13 \times 256$ Conv 3×3
Volume: Operations:	$13 \times 13 \times 384$ Conv 3×3	$13 \times 13 \times 512$ Conv 3×3
Volume: Operations:	$13 \times 13 \times 384$ Conv 3×3	$13 \times 13 \times 1024$ Conv 3×3
Volume: Operations:	$13 \times 13 \times 256$ MP, Fully connect	$13 \times 13 \times 512$ MP, Fully connect
FC6: Operations:	4096 Fully connect	4096 Fully connect
FC7: Operations:	4096 Fully connect	4096 Fully connect
FC8: Operations:	1000 Softmax	1000 Softmax

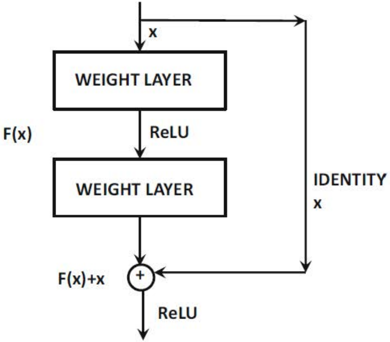
Convolutional Neural Network

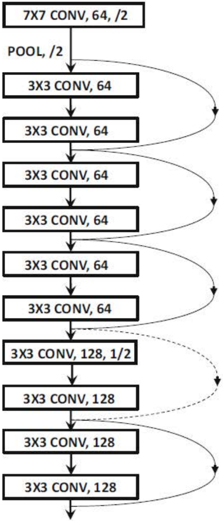
LeNet-5



Convolutional Neural Network

ResNet





Convolutional Neural Network

Name	Year	Number of Layers	Top-5 Error
–	Before 2012	≤ 5	$> 25\%$
AlexNet	2012	8	15.4%
ZfNet/Clarifai	2013	$8/> 8$	14.8% / 11.1%
VGG	2014	19	7.3%
GoogLeNet	2014	22	6.7%
ResNet	2015	152	3.6%

What are the convolution kernels?

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

3x3 Kernel

0	1	0
1	0	1
-1	-1	-1

With a stride=1

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a stride=1

1 x0	4 x1	5 x0	2	4	2	3
3 x1	0 x0	2 x1	1	0	6	1
2x-1	4x-1	2x-1	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

1				

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a stride=1

1	4 x0	5 x1	2 x0	4	2	3
3	0 x1	2 x0	1 x1	0	6	1
2	4x-1	2x-1	3x-1	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

1	-3			

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a stride=1

1	4 x0	5 x1	2 x0	4	2	3
3	0 x1	2 x0	1 x1	0	6	1
2	4x-1	2x-1	3x-1	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

1	-3			

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a stride=1

1	4	5 x0	2 x1	4 x0	2	3
3	0	2 x1	1 x0	0 x1	6	1
2	4	2x-1	3x-1	4x-1	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

1	-3	-5		

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a **stride=1**

1	4	5	2 x0	4 x1	2 x0	3
3	0	2	1 x1	0 x0	6 x1	1
2	4	2	3x-1	4x-1	0x-1	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

1	-3	-5	4	

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a stride=1

1	4	5	2	4 x0	2 x1	3 x0
3	0	2	1	0 x1	6 x0	1 x1
2	4	2	3	4x-1	0x-1	2x-1
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

1	-3	-5	4	-3

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a stride=1

1	4	5	2	4	2	3
3 x0	0 x1	2 x0	1	0	6	1
2 x1	4 x0	2 x1	3	4	0	2
3x-1	4x-1	1x-1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

1	-3	-5	4	-3
-4				

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a stride=1

1	4	5	2	4	2	3
3	0 x0	2 x1	1 x0	0	6	1
2	4 x1	2 x0	3 x1	4	0	2
3	4x-1	1x-1	4x-1	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

1	-3	-5	4	-3
-4	0			

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a stride=1

1	4	5	2	4	2	3
3	0	2 x0	1 x1	0 x0	6	1
2	4	2 x1	3 x0	4 x1	0	2
3	4	1x-1	4x-1	6x-1	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

1	-3	-5	4	-3
-4	0	-4		

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a stride=1

1	4	5	2	4	2	3
3	0	2	1 x0	0 x1	6 x0	1
2	4	2	3 x1	4 x0	0 x1	2
3	4	1	4x-1	6x-1	5x-1	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

1	-3	-5	4	-3
-4	0	-4	-4	

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a stride=1

1	4	5	2	4	2	3
3	0	2	1	0 x0	6 x1	1 x0
2	4	2	3	4 x1	0 x0	2 x1
3	4	1	4	6x-1	5x-1	2x-1
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

1	-3	-5	4	-3
-4	0	-4	-4	-1

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a stride=1

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2 x0	2 x1	4 x0
4	2	5	5	4 x1	3 x0	1 x1
1	4	0	1	1x-1	0x-1	6x-1

1	-3	-5	4	-3
-4	0	-4	-4	-1
4	6	4	7	0
-3	-9	-6	-2	3
4	4	9	8	0

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

3x3 Kernel

0	1	0
1	0	1
-1	-1	-1

With a stride=1

7

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

7

1	-3	-5	4	-3
-4	0	-4	-4	-1
4	6	4	7	0
-3	-9	-6	-2	3
4	4	9	8	0

5

5

$$Output\ size = \frac{7-3}{1} + 1 = 5$$

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a **stride=2**

1 x0	4 x1	5 x0	2	4	2	3
3 x1	0 x0	2 x1	1	0	6	1
2x-1	4x-1	2x-1	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

1		

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a **stride=2**

1	4	5 x0	2 x1	4 x0	2	3
3	0	2 x1	1 x0	0 x1	6	1
2	4	2x-1	3x-1	4x-1	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

1	-5	

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a stride=2

1	4	5	2	4 x0	2 x1	3 x0
3	0	2	1	0 x1	6 x0	1 x1
2	4	2	3	4x-1	0x-1	2x-1
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

1	-5	-3

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a stride=2

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2 x0	4 x1	2 x0	3	4	0	2
3 x1	4 x0	1 x1	4	6	5	2
2x-1	0x-1	2x-1	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

1	-5	-3
4		

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a **stride=2**

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2 x0	3 x1	4 x0	0	2
3	4	1 x1	4 x0	6 x1	5	2
2	0	2x-1	2x-1	2x-1	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

1	-5	-3
4	4	

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a stride=2

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4 x0	0 x1	2 x0
3	4	1	4	6 x1	5 x0	2 x1
2	0	2	2	2x-1	2x-1	4x-1
4	2	5	5	4	3	1
1	4	0	1	1	0	6

1	-5	-3
4	4	0

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a **stride=2**

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2 x0	0 x1	2 x0	2	2	2	4
4 x1	2 x0	5 x1	5	4	3	1
1x-1	4x-1	0x-1	1	1	0	6

1	-5	-3
4	4	0
4		

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a stride=2

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2	0	2 x0	2 x1	2 x0	2	4
4	2	5 x1	5 x0	4 x1	3	1
1	4	0x-1	1x-1	1x-1	0	6

1	-5	-3
4	4	0
4	9	

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

With a **stride=2**

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2 x0	2 x1	4 x0
4	2	5	5	4 x1	3 x0	1 x1
1	4	0	1	1x-1	0x-1	6x-1

1	-5	-3
4	4	0
4	9	0

Convolutional Neural Network

A kernel is a weights matrix that applies to cells and aggregate into a sum resulting in a convolution like in

3x3 Kernel

0	1	0
1	0	1
-1	-1	-1

With a stride=2

7 × 7

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

3 × 3

1	-5	-3
4	4	0
4	9	0

$$Output\ size = \frac{7-3}{2} + 1 = 3$$

Convolutional Neural Network

What about stride 3?

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

Convolutional Neural Network

What about stride 3? Cannot be.

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

Convolutional Neural Network

Zero-padding for border

3x3 Kernel

0	1	0
1	0	1
-1	-1	-1

With a stride=1

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

Convolutional Neural Network

Zero-padding for border

3x3 Kernel

0	1	0
1	0	1
-1	-1	-1

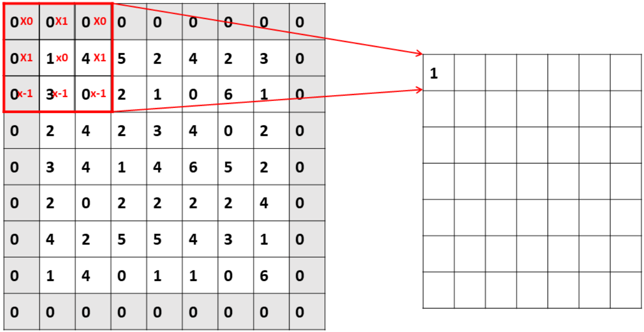
With a stride=1

0	0	0	0	0	0	0	0	0
0	1	4	5	2	4	2	3	0
0	3	0	2	1	0	6	1	0
0	2	4	2	3	4	0	2	0
0	3	4	1	4	6	5	2	0
0	2	0	2	2	2	2	4	0
0	4	2	5	5	4	3	1	0
0	1	4	0	1	1	0	6	0
0	0	0	0	0	0	0	0	0

Convolutional Neural Network

Zero-padding for border

With a **stride=1**

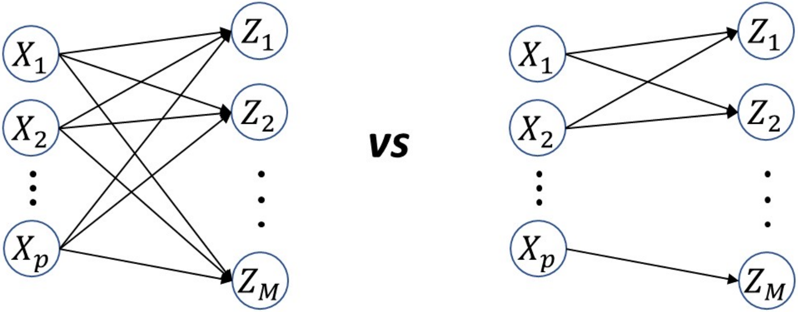


Complete – DIY

Convolutional Neural Network

Benefits of CNN:

- Sparse connectivity: for instance, compare 224x224x3=150528 connected to each other through weighted links *vs.* 11x11x3=363 weights in a kernel



Convolutional Neural Network

Benefits of CNN:

- Translation invariance: applying a convolution kernel with stride 1 will result in the same outcome for a translated image

1	4	5				
3	0	2				
2	4	2				

			1	4	5	
			3	0	2	
			2	4	2	

Convolutional Neural Network

Application of kernels: Sobel's Edge Detection (first transform color images into grey images)

Apply them directional gradients separately …

$G_x=$

-1	0	1
-2	0	2
-1	0	1

horizontally

$G_y=$

1	2	1
0	0	0
-1	-2	-1

vertically

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

… then add up the outcomes.

Convolutional Neural Network

Application of kernels: Sobel's Edge Detection (first transform color images into grey images)

What are the sub-sampling layers or pooling kernels?

Convolutional Neural Network

Sub-sampling layers are non trainable using some pooling kernels. A pooling kernel applies to cells and aggregate them into a maximum (MP) or an average (AP) resulting in a convolution like in

With a stride=2

Max

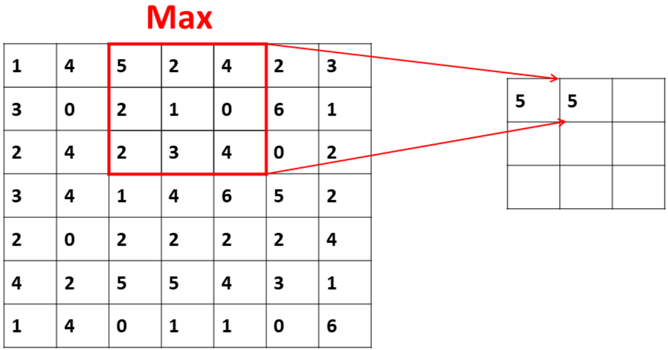
1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

5		

Convolutional Neural Network

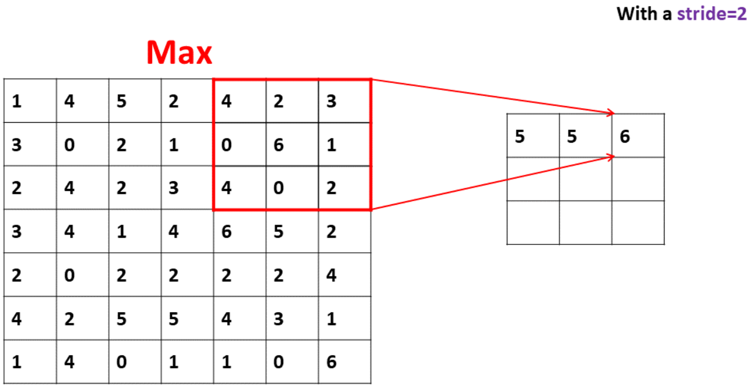
Sub-sampling layers are non trainable using some pooling kernels. A pooling kernel applies to cells and aggregate them into a maximum (MP) or an average (AP) resulting in a convolution like in

With a **stride=2**



Convolutional Neural Network

Sub-sampling layers are non trainable using some pooling kernels. A pooling kernel applies to cells and aggregate them into a maximum (MP) or an average (AP) resulting in a convolution like in



Convolutional Neural Network

Sub-sampling layers are non trainable using some pooling kernels. A pooling kernel applies to cells and aggregate them into a maximum (MP) or an average (AP) resulting in a convolution like in

With a stride=2

Max

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

5	5	6
4		

Convolutional Neural Network

Sub-sampling layers are non trainable using some pooling kernels. A pooling kernel applies to cells and aggregate them into a maximum (MP) or an average (AP) resulting in a convolution like in

With a stride=2

Max

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

5	5	6
4	6	

Convolutional Neural Network

Sub-sampling layers are non trainable using some pooling kernels. A pooling kernel applies to cells and aggregate them into a maximum (MP) or an average (AP) resulting in a convolution like in

With a stride=2

Max

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

5	5	6
4	6	6

Convolutional Neural Network

Sub-sampling layers are non trainable using some pooling kernels. A pooling kernel applies to cells and aggregate them into a maximum (MP) or an average (AP) resulting in a convolution like in

With a stride=2

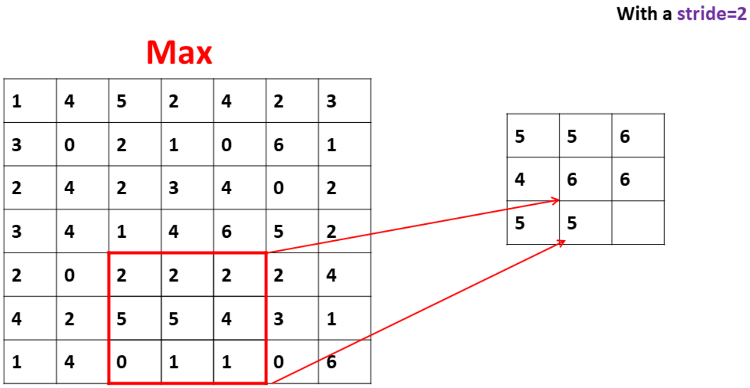
Max

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

5	5	6
4	6	6
5		

Convolutional Neural Network

Sub-sampling layers are non trainable using some pooling kernels. A pooling kernel applies to cells and aggregate them into a maximum (MP) or an average (AP) resulting in a convolution like in



Convolutional Neural Network

Sub-sampling layers are non trainable using some pooling kernels. A pooling kernel applies to cells and aggregate them into a maximum (MP) or an average (AP) resulting in a convolution like in

With a **stride=2**

Max

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

5	5	6
4	6	6
5	5	6

Convolutional Neural Network

Sub-sampling layers are non trainable using some pooling kernels. A pooling kernel applies to cells and aggregate them into a maximum (MP) or an average (AP – Complete it – DIY) resulting in a convolution like in

With a stride=2

Average

1	4	5	2	4	2	3
3	0	2	1	0	6	1
2	4	2	3	4	0	2
3	4	1	4	6	5	2
2	0	2	2	2	2	4
4	2	5	5	4	3	1
1	4	0	1	1	0	6

?		

Convolutional Neural Network

A sub-sampling layer makes images smaller with fewer parameters that doesn't alter the image

Convolutional Neural Network

Application of kernels: Image blurring

Convolutional Neural Network

Example (Hand gestures-Sahoo et al. 2021)

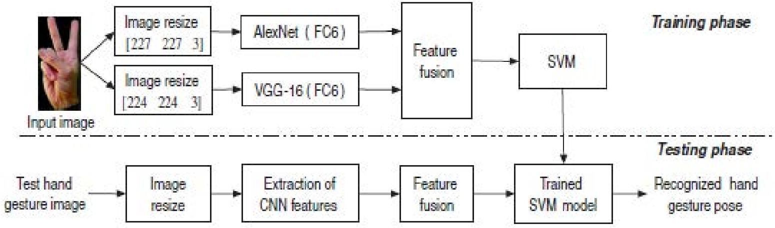


FIG. 13.4

Block diagram representation of the proposed hand gesture recognition system.

Convolutional Neural Network

Example (Hand gestures-Sahoo et al. 2021)

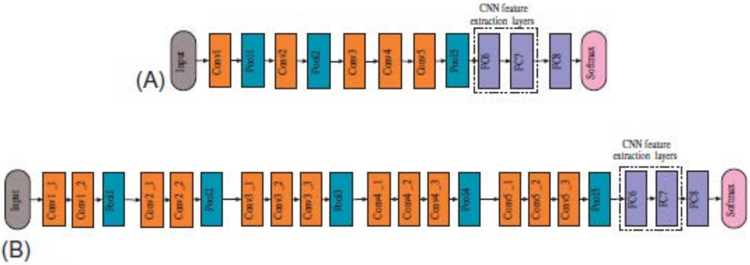


FIG. 13.5

Architecture of pretrained CNNs used in this work. The CNN features are extracted from the fully connected layers such as FC6 and FC7 from the pretrained CNN model, which is shown in the *dotted mark*. (A) AlexNet and (B) VGG-16.

End of Set 8