18-661 Introduction to Machine Learning

Neural Networks-III

Spring 2020

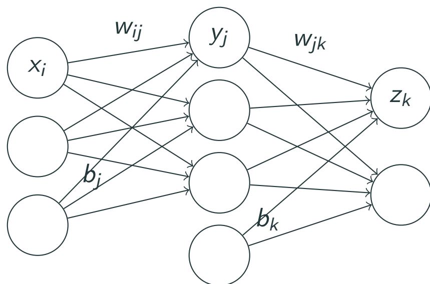
ECE – Carnegie Mellon University

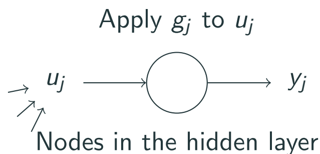
Outline

1. Review: Inference using a Trained Network: Forward Propagation
2. Review: Training a Neural Network: Backpropagatiopn
3. Optimizing SGD Parameters for Faster Convergence
4. Universality and Depth
5. Deep Neural Networks (DNNs)

Review: Inference using a Trained Network: Forward Propagation

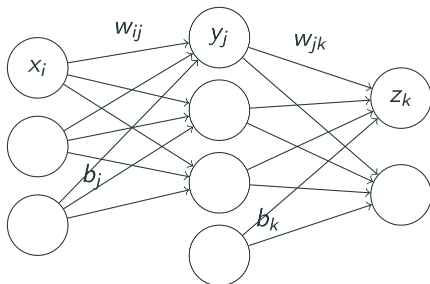
How do you perform inference using a trained neural network?

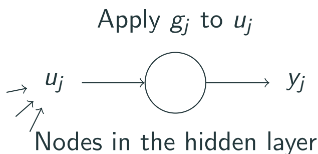




- Expressing outputs z in terms of inputs x is called forward-propagation.
  - Express inputs u_j to the hidden layer in terms of x

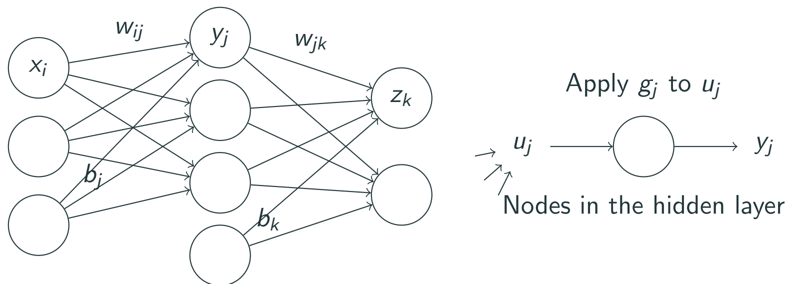
How do you perform inference using a trained neural network?





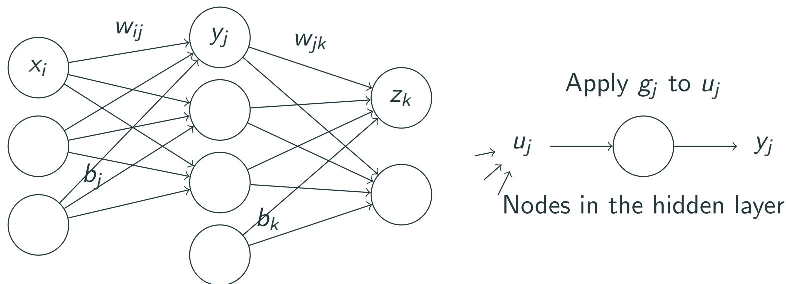
- Expressing outputs z in terms of inputs x is called forward-propagation.
  - Express inputs u_j to the hidden layer in terms of x: $u_j = \sum_i w_{ij} x_i + b_j$
  - Express outputs y_j of the hidden layer in terms of x

How do you perform inference using a trained neural network?



- Expressing outputs z in terms of inputs x is called forward-propagation.
  - Express inputs u_j to the hidden layer in terms of x: $u_j = \sum_i w_{ij} x_i + b_j$
  - Express outputs y_j of the hidden layer in terms of x: $y_j = g(\sum_i w_{ij} x_i + b_j)$
  - Express inputs to the final layer in terms of x
  - Express outputs z_k of the final layer in terms of x

How do you perform inference using a trained neural network?



- Expressing outputs z in terms of inputs x is called forward-propagation.
  - Express inputs u_j to the hidden layer in terms of x: $u_j = \sum_i w_{ij} x_i + b_j$
  - Express outputs y_j of the hidden layer in terms of x: $y_j = g(\sum_i w_{ij} x_i + b_j)$
  - Express inputs to the final layer in terms of x
  - Express outputs z_k of the final layer in terms of x: $z_k = g(\sum_j w_{jk} y_j + b_k)$

Review: Training a Neural Network: Backpropagatiopn

Learning Parameters

How to learn the parameters?

- Choose the right loss function
  - Regression: Least-square loss (in class)

$$min \sum_n (f(\mathbf{x}_n) - t_n)^2$$

Learning Parameters

How to learn the parameters?

- Choose the right loss function
  - Regression: Least-square loss (in class)

$$min \sum_n (f(\mathbf{x}_n) - t_n)^2$$

  - Classification: cross-entropy loss (in the homework)

$$min - \sum_n \sum_k t_{nk} \log f_k(\mathbf{x}_n) + (1 - t_{nk}) \log(1 - f_k(\mathbf{x}_n))$$

- Hard optimization problem because f (the output of the neural network) is a complicated function of $\mathbf{x}_n$

Learning Parameters

How to learn the parameters?

- Choose the right loss function
  - Regression: Least-square loss (in class)

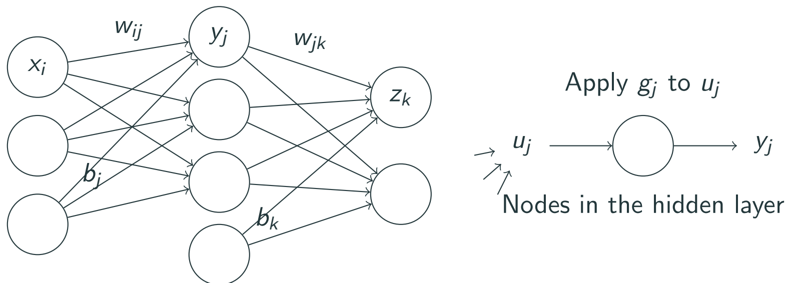
$$min \sum_n (f(\mathbf{x}_n) - t_n)^2$$

  - Classification: cross-entropy loss (in the homework)

$$min - \sum_n \sum_k t_{nk} \log f_k(\mathbf{x}_n) + (1 - t_{nk}) \log(1 - f_k(\mathbf{x}_n))$$

- Hard optimization problem because f (the output of the neural network) is a complicated function of $\mathbf{x}_n$
  - Solution: Use Stochastic gradient descent (SGD)
  - Many optimization tricks are applied to speed-up SGD convergence

Stochastic gradient descent



- Randomly pick a data point $(\mathbf{x}_n, t_n)$
- Compute the gradient using only this data point, for example,

$$\Delta = \frac{\partial [f(\mathbf{x}_n) - t_n]^2}{\partial w}$$

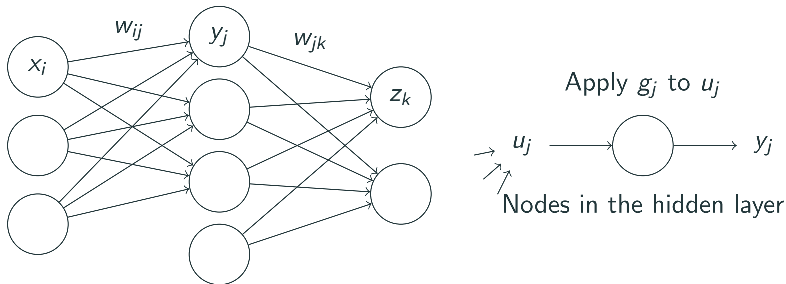
- Update the parameters: $\mathbf{w} \leftarrow \mathbf{w} - \eta \Delta$
- Iterate the process until some (pre-specified) stopping criteria

Updating the parameter values

Back-propagate the error. Given parameters w, b:

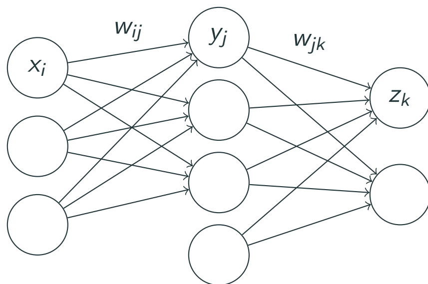
- Step 1: Forward-propagate to find z_k in terms of the input (the "feed-forward signals").
- Step 2: Calculate output error E by comparing the predicted output z_k to its true value t_k.
- Step 3: Back-propagate E by weighting it by the gradients of the associated activation functions and the weights in previous layers.
- Step 4: Calculate the gradients $\frac{\partial E}{\partial w}$ and $\frac{\partial E}{\partial b}$ for the parameters w, b at each layer based on the backpropagated error signal and the feedforward signals from the inputs.
- Step 5: Update the parameters using the calculated gradients $w \leftarrow w - \eta \frac{\partial E}{\partial w}$, $b \leftarrow b - \eta \frac{\partial E}{\partial b}$ where η is the step size.

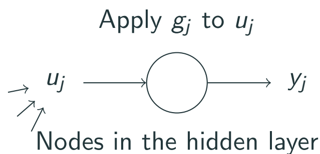
Illustrative example



- w_{ij}: weights connecting node i in layer $(\ell - 1)$ to node j in layer ℓ.
- b_j, b_k: bias for nodes j and k.
- u_j, u_k: inputs to nodes j and k (where $u_j = b_j + \sum_i x_i w_{ij}$).
- g_j, g_k: activation function for node j (applied to u_j) and node k.
- $y_j = g_j(u_j)$, $z_k = g_k(u_k)$: output/activation of nodes j and k.
- t_k: target value for node k in the output layer.

Illustrative example (steps 1 and 2)

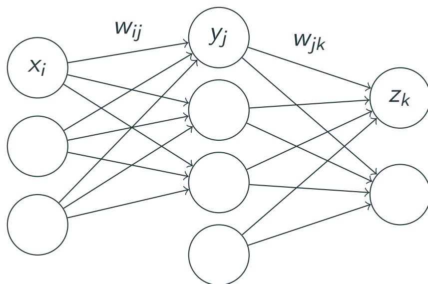


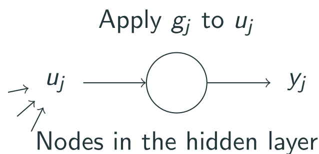


- Step 1: Forward-propagate for each output z_k.

$$z_k = g_k(u_k) = g_k(b_k + \sum_j y_j w_{jk}) = g_k(b_k + \sum_j g_j(b_j + \sum_i x_i w_{ij}) w_{jk})$$

Illustrative example (steps 1 and 2)



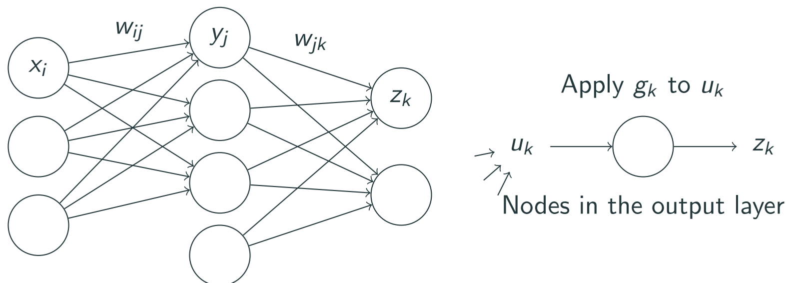


- Step 1: Forward-propagate for each output z_k.

$$z_k = g_k(u_k) = g_k(b_k + \sum_j y_j w_{jk}) = g_k(b_k + \sum_j g_j(b_j + \sum_i x_i w_{ij})w_{jk})$$

- Step 2: Find the error. Let's assume that the error function is the sum of the squared differences between the target values t_k and the network output z_k: $E = \frac{1}{2}\sum_{k \in K}(z_k - t_k)^2$.

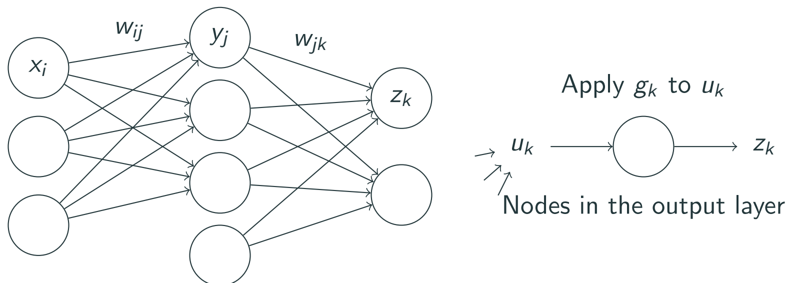
Illustrative example (step 3, output layer)



Step 3: Backpropagate the error. Let's start at the output layer with weight w_{jk}, recalling that $E = \frac{1}{2}\sum_{k \in K}(z_k - t_k)^2$, $u_k = b_k + \sum_j w_{jk} y_j$:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial u_k}\frac{\partial u_k}{\partial w_{jk}} = (z_k - t_k)\frac{\partial z_k}{\partial u_k}\frac{\partial u_k}{\partial w_{jk}}$$

Illustrative example (step 3, output layer)

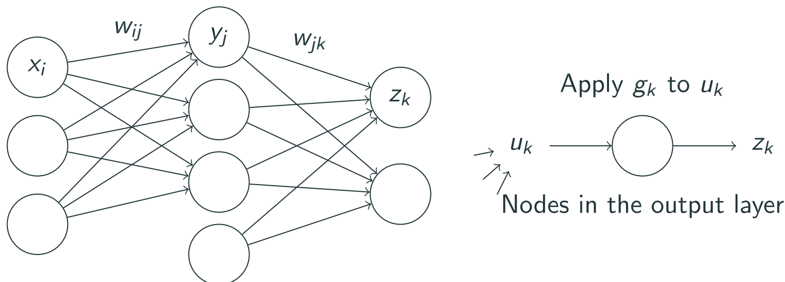


Step 3: Backpropagate the error. Let's start at the output layer with weight w_{jk}, recalling that $E = \frac{1}{2}\sum_{k \in K}(z_k - t_k)^2$, $u_k = b_k + \sum_j w_{jk} y_j$:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial u_k}\frac{\partial u_k}{\partial w_{jk}} = (z_k - t_k)\frac{\partial z_k}{\partial u_k}\frac{\partial u_k}{\partial w_{jk}}$$

$$= (z_k - t_k) g_k'(u_k) \frac{\partial}{\partial w_{jk}} u_k$$

Illustrative example (step 3, output layer)



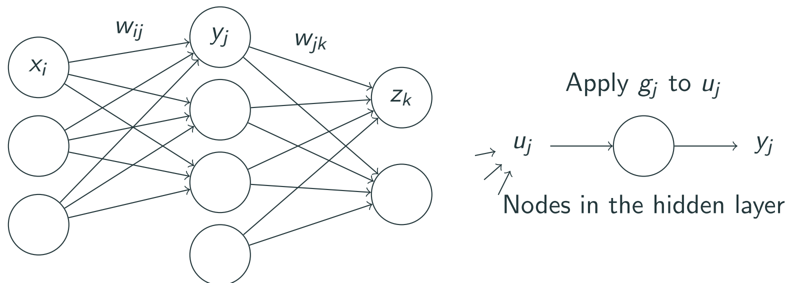
Step 3: Backpropagate the error. Let's start at the output layer with weight w_{jk}, recalling that $E = \frac{1}{2}\sum_{k \in K}(z_k - t_k)^2$, $u_k = b_k + \sum_j w_{jk} y_j$:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial u_k}\frac{\partial u_k}{\partial w_{jk}} = (z_k - t_k)\frac{\partial z_k}{\partial u_k}\frac{\partial u_k}{\partial w_{jk}}$$

$$= (z_k - t_k) g_k'(u_k) \frac{\partial}{\partial w_{jk}} u_k = (z_k - t_k) g_k'(u_k) y_j = \delta_k y_j$$

where $\delta_k = (z_k - t_k) g_k'(u_k)$ is called the error in u_k.

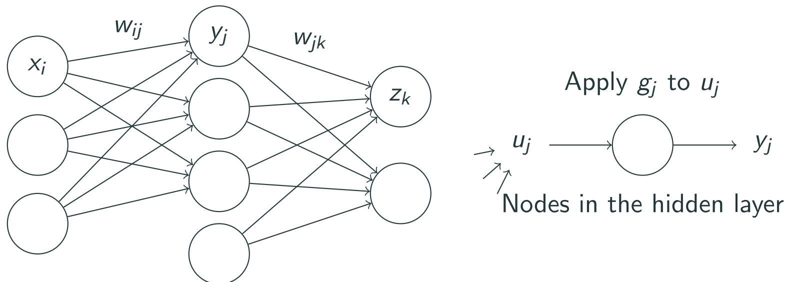
Illustrative example (step 3, hidden layer)



Step 3 (cont'd): Now let's consider w_{ij} in the hidden layer, recalling $u_j = b_i + \sum_i x_i w_{ij}$, $u_k = b_k + \sum_j g_j(u_j) w_{jk}$, $z_k = g_k(u_k)$:

$$\frac{\partial E}{\partial w_{ij}} = \sum_{k \in K} \frac{\partial E}{\partial u_k} \frac{\partial u_k}{\partial y_j} \frac{\partial y_j}{\partial u_j} \frac{\partial u_j}{\partial w_{ij}} = \sum_{k \in K} \delta_k w_{jk} \frac{\partial y_j}{\partial u_j} \frac{\partial u_j}{\partial w_{ij}}$$

Illustrative example (step 3, hidden layer)



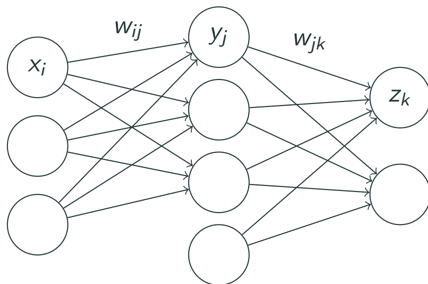
Step 3 (cont'd): Now let's consider w_{ij} in the hidden layer, recalling $u_j = b_i + \sum_i x_i w_{ij}$, $u_k = b_k + \sum_j g_j(u_j) w_{jk}$, $z_k = g_k(u_k)$:

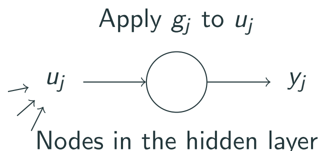
$$\frac{\partial E}{\partial w_{ij}} = \sum_{k \in K} \frac{\partial E}{\partial u_k} \frac{\partial u_k}{\partial y_j} \frac{\partial y_j}{\partial u_j} \frac{\partial u_j}{\partial w_{ij}} = \sum_{k \in K} \delta_k w_{jk} \frac{\partial y_j}{\partial u_j} \frac{\partial u_j}{\partial w_{ij}}$$

$$= \sum_{k \in K} \delta_k w_{jk} g_j'(u_j) x_i = \delta_j x_i$$

where we substituted $\delta_j = g_j'(u_j) \sum_{k \in K} (z_k - t_k) g_k'(u_k) w_{jk}$, the error in u_j.

Illustrative example (steps 3 and 4)



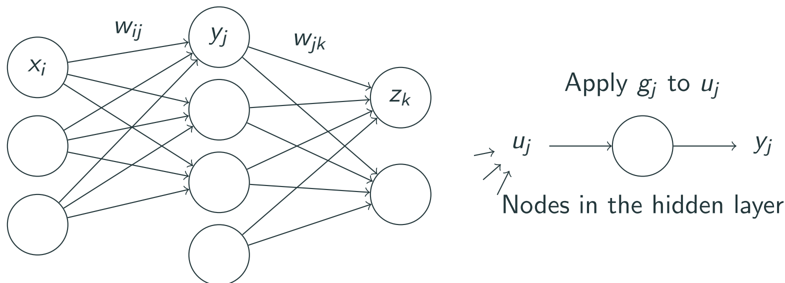


- Step 3 (cont'd): We similarly find that $\frac{\partial E}{\partial b_k} = \delta_k$, $\frac{\partial E}{\partial b_j} = \delta_j$.
- Step 4: Calculate the gradients. We have found that

$$\frac{\partial E}{\partial w_{ij}} = \delta_j x_i \text{ and } \frac{\partial E}{\partial w_{jk}} = \delta_k y_j.$$

where $\delta_k = (z_k - t_k)g_k'(u_k)$, $\delta_j = g_j'(u_j) \sum_{k \in K}(z_k - t_k)g_k'(u_k)w_{jk}$. Now since we know the z_k, y_j, x_i, u_k and u_j for a given set of parameter values w, b, we can use these expressions to calculate the gradients at each iteration and update them.

Illustrative example (steps 4 and 5)



- Step 4: Calculate the gradients. We have found that

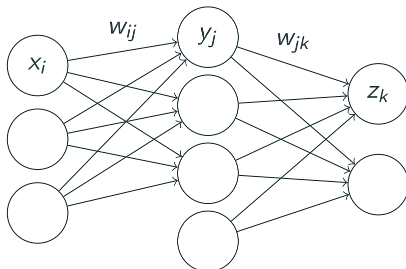
$$\frac{\partial E}{\partial w_{ij}} = \delta_j x_i \text{ and } \frac{\partial E}{\partial w_{jk}} = \delta_k y_j.$$

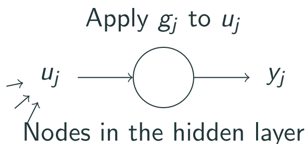
  where $\delta_k = (z_k - t_k)g_k'(u_k)$, $\delta_j = g_j'(u_j)\sum_{k \in K}(z_k - t_k)g_k'(u_k)w_{jk}$.

- Step 5: Update the weights and biases with learning rate η. For example

$$w_{jk} \leftarrow w_{jk} - \eta\frac{\partial E}{\partial w_{jk}} \text{ and } w_{ij} \leftarrow w_{ij} - \eta\frac{\partial E}{\partial w_{ij}}$$

High-level Procedure: Can be Used with More Hidden Layers





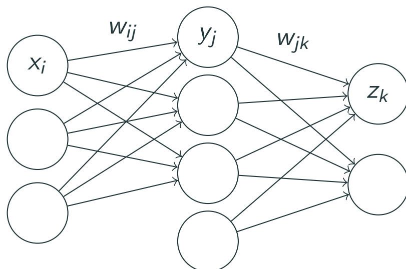
Final Layer

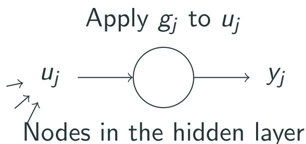
- Error in each of its outputs is $z_k - t_k$.
- Error in input u_k to the final layer is $\delta_k = g_k'(u_k)(z_k - t_k)$

Hidden Layer

- Error in output y_j is $\sum_{k \in K} \delta_k w_{jk}$.
- Error in the input u_j of the hidden layer is $\delta_j = g_j'(u_j) \sum_{k \in K} \delta_k w_{jk}$

High-level Procedure: Can be Used with More Hidden Layers





Final Layer

- Error in each of its outputs is $z_k - t_k$.
- Error in input u_k to the final layer is $\delta_k = g_k'(u_k)(z_k - t_k)$

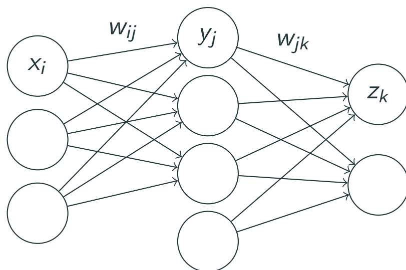
Hidden Layer

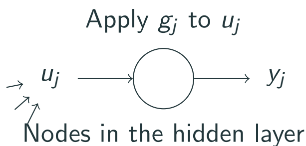
- Error in output y_j is $\sum_{k \in K} \delta_k w_{jk}$.
- Error in the input u_j of the hidden layer is $\delta_j = g_j'(u_j) \sum_{k \in K} \delta_k w_{jk}$

The gradient w.r.t. w_{ij} is $x_i \delta_j$.

Vectorized Implementation

Much faster than implementing a loop over all neurons in each layer



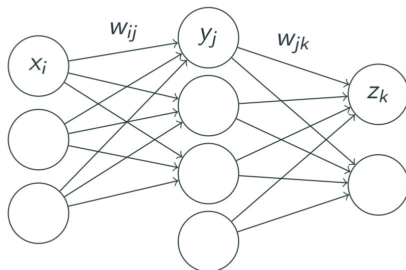


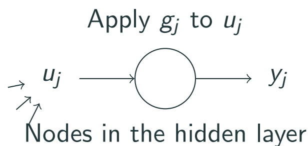
Forward-Propagation

- Represent the weights between layers $l-1$ and l as a matrix $\mathbf{W}^{(l)}$ and biases as a row vector $\mathbf{b}^{(l)}$

Vectorized Implementation

Much faster than implementing a loop over all neurons in each layer



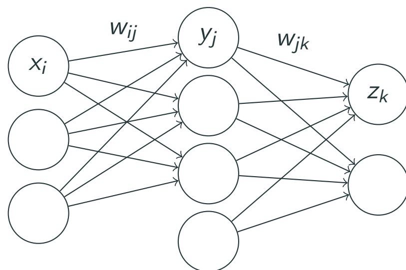


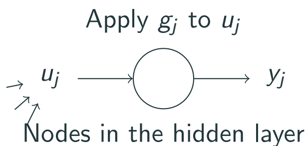
Forward-Propagation

- Represent the weights between layers $l-1$ and l as a matrix $\mathbf{W}^{(l)}$ and biases as a row vector $\mathbf{b}^{(l)}$
- Outputs of layer $l-1$ are in a row vector $\mathbf{y}^{(l-1)}$. Then we have $\mathbf{u}^{(l)} = \mathbf{y}^{(l-1)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)}$.

Vectorized Implementation

Much faster than implementing a loop over all neurons in each layer



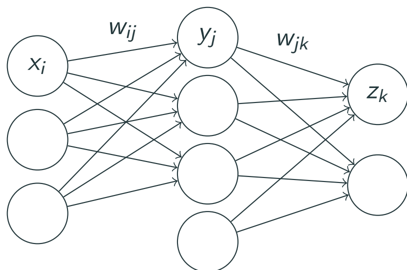


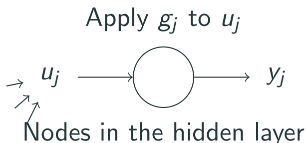
Forward-Propagation

- Represent the weights between layers $l-1$ and l as a matrix $\mathbf{W}^{(l)}$ and biases as a row vector $\mathbf{b}^{(l)}$
- Outputs of layer $l-1$ are in a row vector $\mathbf{y}^{(l-1)}$. Then we have $\mathbf{u}^{(l)} = \mathbf{y}^{(l-1)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)}$.
- Outputs of layer l are in the row vector $\mathbf{y}^{(l)} = g(\mathbf{u}^{(l)})$.

Vectorized Implementation

Much faster than implementing a loop over all neurons in each layer



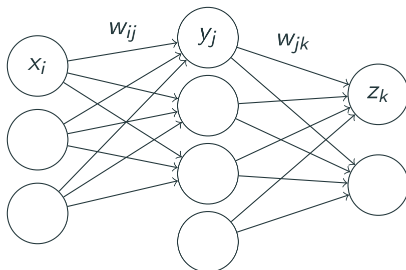


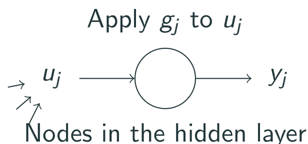
Back-Propagation

- For each layer l find $\Delta^{(l)}$, the vector of errors in $\mathbf{u}^{(l)}$ in terms of the final error

Vectorized Implementation

Much faster than implementing a loop over all neurons in each layer



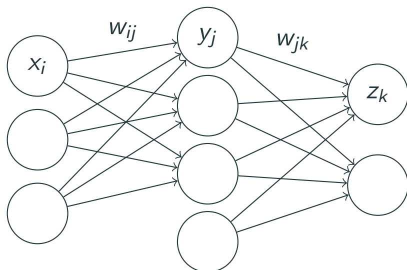


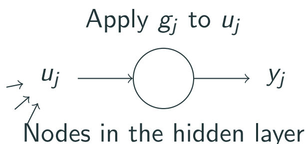
Back-Propagation

- For each layer l find $\Delta^{(l)}$, the vector of errors in $\mathbf{u}^{(l)}$ in terms of the final error
- Update weights $\mathbf{W}^{(l)}$ using $\Delta^{(l)}$

Vectorized Implementation

Much faster than implementing a loop over all neurons in each layer





Back-Propagation

- For each layer l find $\Delta^{(l)}$, the vector of errors in $\mathbf{u}^{(l)}$ in terms of the final error
- Update weights $\mathbf{W}^{(l)}$ using $\Delta^{(l)}$
- Recursively find $\Delta^{(l-1)}$ in terms $\Delta^{(l)}$

Optimizing SGD Parameters for Faster Convergence

Mini-batch SGD

- Recall the empirical risk loss function that we considered for the backpropagation discussion

$$E = \sum_{n=1}^{N} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

Mini-batch SGD

- Recall the empirical risk loss function that we considered for the backpropagation discussion

$$E = \sum_{n=1}^{N} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

- For large training datasets (large N), then computing gradients with respect to each datapoint is expensive. For example, for the last year, the batch gradients are

$$\frac{\partial E}{\partial w_{jk}} = \sum_{n=1}^{N}(z_k - t_k)$$

Mini-batch SGD

- Recall the empirical risk loss function that we considered for the backpropagation discussion

$$E = \sum_{n=1}^{N} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

- For large training datasets (large N), then computing gradients with respect to each datapoint is expensive. For example, for the last year, the batch gradients are

$$\frac{\partial E}{\partial w_{jk}} = \sum_{n=1}^{N}(z_k - t_k)$$

- Therefore we use stochastic gradient descent (SGD), where we choose a random data point $\mathbf{x}_n$ and use $E = \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$ instead of the entire sum

Mini-batch SGD

- Mini-batch SGD is in between these two extremes

Mini-batch SGD

- Mini-batch SGD is in between these two extremes
- In each iteration, we choose a set S of m samples from the N training samples and use

$$E = \sum_{n \in S} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

for backpropagation

Mini-batch SGD

- Mini-batch SGD is in between these two extremes
- In each iteration, we choose a set S of m samples from the N training samples and use

$$E = \sum_{n \in S} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

for backpropagation

- Small m saves per-iteration computing cost, but increases noise in the gradients and yields worse error convergence

Mini-batch SGD

- Mini-batch SGD is in between these two extremes
- In each iteration, we choose a set S of m samples from the N training samples and use

$$E = \sum_{n \in S} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

  for backpropagation
- Small m saves per-iteration computing cost, but increases noise in the gradients and yields worse error convergence
- Large m reduces gradient noise and gives better error convergence, but increases computing cost per iteration

Mini-batch SGD

- Mini-batch SGD is in between these two extremes

Mini-batch SGD

- Mini-batch SGD is in between these two extremes
- In each iteration, we choose a set S of m samples from the N training samples and use

$$E = \sum_{n \in S} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

for backpropagation

Mini-batch SGD

- Mini-batch SGD is in between these two extremes
- In each iteration, we choose a set S of m samples from the N training samples and use

$$E = \sum_{n \in S} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

  for backpropagation
- Small m saves per-iteration computing cost, but increases noise in the gradients and yields worse error convergence

Mini-batch SGD

- Mini-batch SGD is in between these two extremes
- In each iteration, we choose a set S of m samples from the N training samples and use

$$E = \sum_{n \in S} \frac{1}{2}(f(\mathbf{x}_n) - t_n)^2$$

  for backpropagation
- Small m saves per-iteration computing cost, but increases noise in the gradients and yields worse error convergence
- Large m reduces gradient noise and typically gives better error convergence, but increases computing cost per iteration

How to Choose Mini-batch size m

- Small training datasets – use batch gradient descent $m = N$
- Large training datasets – typical m are 64, 128, 256 ... whatever fits in the CPU/GPU memory
- Mini-batch size is another hyperparameter that you have to tune

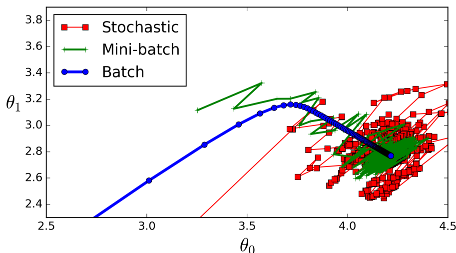
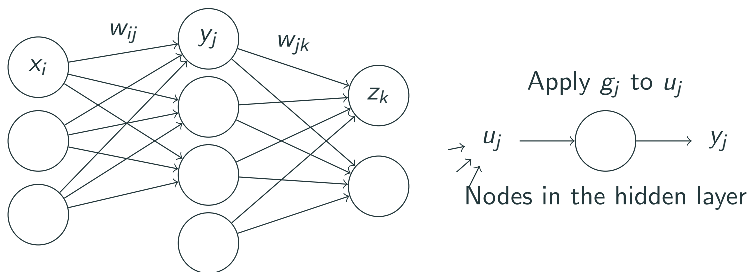


Image source: `https://github.com/buomsoo-kim/Machine-learning-toolkits-with-python`

Vectorized Implementation of Mini-batch SGD

Much faster than implementing a loop over all neurons in each layer and all samples in a mini-batch

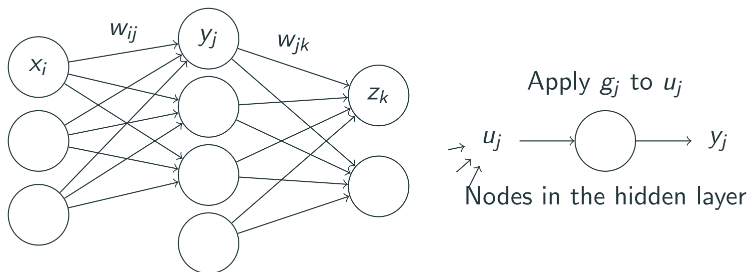


Forward-Propagation

- Represent the weights between layers $l-1$ and l as a matrix $\mathbf{W}^{(l)}$ and biases as a vector $\mathbf{b}^{(l)}$ (dimensions do NOT depend on m)

Vectorized Implementation of Mini-batch SGD

Much faster than implementing a loop over all neurons in each layer and all samples in a mini-batch

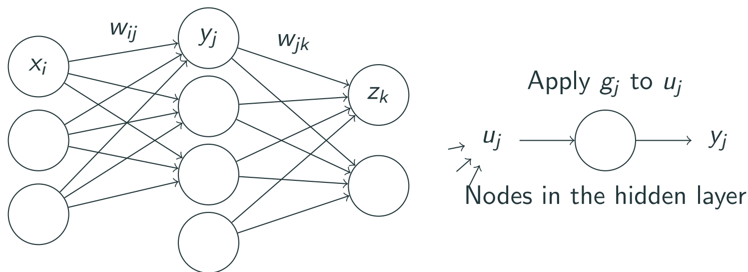


Forward-Propagation

- Represent the weights between layers $l-1$ and l as a matrix $\mathbf{W}^{(l)}$ and biases as a vector $\mathbf{b}^{(l)}$ (dimensions do NOT depend on m)
- Outputs of layer $l-1$ are arranged in an $m \times n_{l-l}$ size matrix $\mathbf{Y}^{(l-1)}$, where each row is the layer $l-1$ outputs for one sample.

Vectorized Implementation of Mini-batch SGD

Much faster than implementing a loop over all neurons in each layer and all samples in a mini-batch

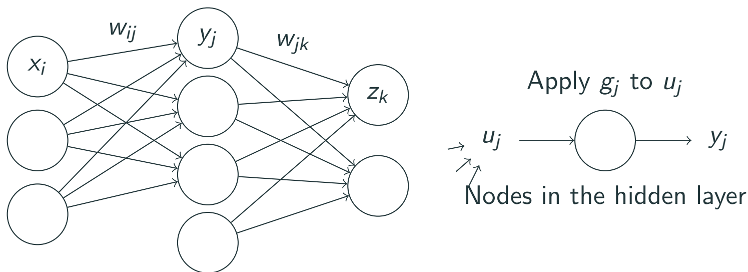


Forward-Propagation (contd)

- Then we have $m \times n_l$ matrix $\mathbf{U}^{(l)} = \mathbf{Y}^{(l-1)}\mathbf{W}^{(l)} + [1, 1, \ldots, 1]^T \mathbf{b}^{(l)}$ of layer l inputs, where $[1, 1, \ldots, 1]^T$ is a column of m ones

Vectorized Implementation of Mini-batch SGD

Much faster than implementing a loop over all neurons in each layer and all samples in a mini-batch

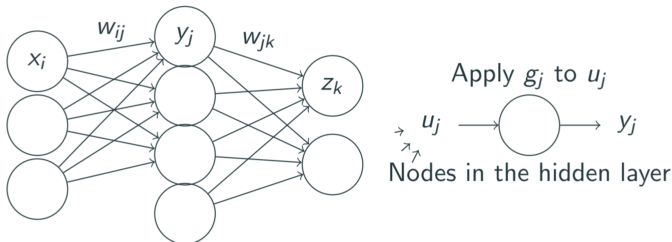


Forward-Propagation (contd)

- Then we have $m \times n_l$ matrix $\mathbf{U}^{(l)} = \mathbf{Y}^{(l-1)}\mathbf{W}^{(l)} + [1, 1, \ldots, 1]^T\mathbf{b}^{(l)}$ of layer l inputs, where $[1, 1, \ldots, 1]^T$ is a column of m ones
- Outputs of layer l is an $m \times n_{(l)}$ matrix $\mathbf{Y}^{(l)} = g(\mathbf{U}^{(l)})$.

Vectorized Implementation of Mini-batch SGD

Much faster than implementing a loop over all neurons in each layer and all samples in a mini-batch

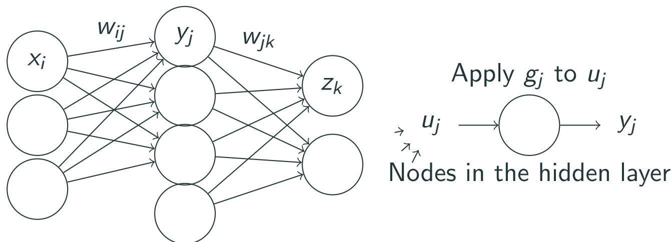


Back-Propagation

- For each layer l find the $m \times n_{(l)}$ size matrix $\Delta^{(l)}$ of errors in $\mathbf{U}^{(l)}$ in terms of the final error E

Vectorized Implementation of Mini-batch SGD

Much faster than implementing a loop over all neurons in each layer and all samples in a mini-batch

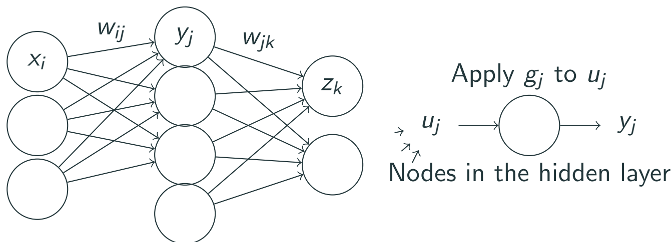


Back-Propagation

- For each layer l find the $m \times n_{(l)}$ size matrix $\Delta^{(l)}$ of errors in $\mathbf{U}^{(l)}$ in terms of the final error E
- Update weights $\mathbf{W}^{(l)}$ using $\Delta^{(l)}$

Vectorized Implementation of Mini-batch SGD

Much faster than implementing a loop over all neurons in each layer and all samples in a mini-batch



Back-Propagation

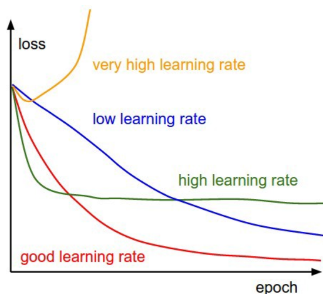
- For each layer l find the $m \times n_{(l)}$ size matrix $\Delta^{(l)}$ of errors in $\mathbf{U}^{(l)}$ in terms of the final error E
- Update weights $\mathbf{W}^{(l)}$ using $\Delta^{(l)}$
- Recursively find $\Delta^{(l-1)}$ in terms $\Delta^{(l)}$

Learning Rate

- SGD Update Rule

$$w^{(t+1)} = w^{(t)} - \eta \frac{\partial E}{\partial w^{(t)}} = w^{(t)} - \eta \nabla E(w^{(t)})$$

- Large η; Faster convergence, but higher error floor (the flat portion of each curve)
- Small η: Slow convergence, but lower error floor (the blue curve will eventually go below the red curve)
- To get the best of both worlds, decay η over time



How to Decay the Learning Rate?

A common way to decay η

- Start with some learning rate, say $\eta = 0.1$

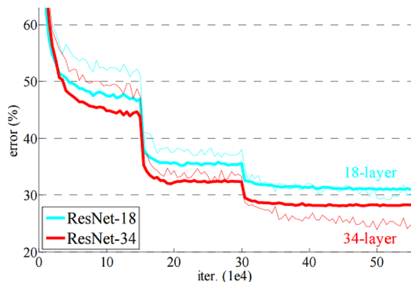


Image Source: `http://www.bdhammel.com/learning-rates/`

How to Decay the Learning Rate?

A common way to decay η

- Start with some learning rate, say $\eta = 0.1$
- Monitor the training loss and wait till it flattens

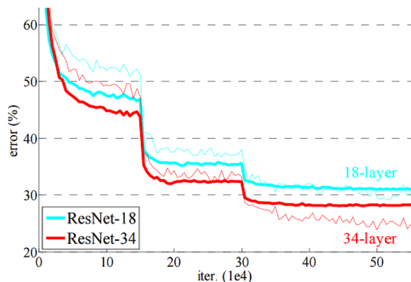


Image Source: `http://www.bdhammel.com/learning-rates/`

How to Decay the Learning Rate?

A common way to decay η

- Start with some learning rate, say $\eta = 0.1$
- Monitor the training loss and wait till it flattens
- Reduce η by a fixed factor, say 5. New $\eta = 0.02$.

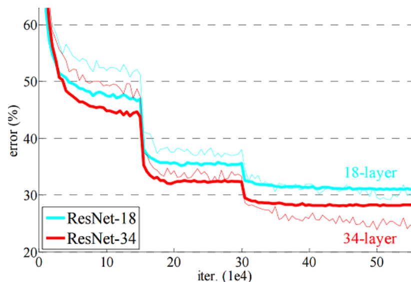


Image Source: `http://www.bdhammel.com/learning-rates/`

How to Decay the Learning Rate?

A common way to decay η

- Start with some learning rate, say $\eta = 0.1$
- Monitor the training loss and wait till it flattens
- Reduce η by a fixed factor, say 5. New $\eta = 0.02$.
- Reduce again by the same factor when curve flattens

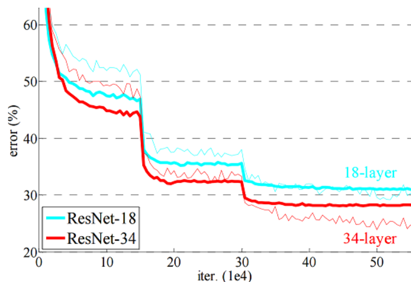


Image Source: `http://www.bdhammel.com/learning-rates/`

How to Decay the Learning Rate?

An alternate approach – AdaGrad [Duchi et al 2011]

- Divide the learning rate η by the square root of the the sum of squares of gradients until that time

How to Decay the Learning Rate?

An alternate approach – AdaGrad [Duchi et al 2011]

- Divide the learning rate η by the square root of the the sum of squares of gradients until that time
- This scaling factor is different for each parameter depending upon the corresponding gradients

$$\mathbf{w}_i^{(t+1)} = w^{(t)} - \frac{\eta}{\sqrt{g_i^{(t)} + \epsilon}} \nabla E(\mathbf{w}_i^{(t)})$$

where $g_i^{(t)} = \sum_{k=1}^{t} (\nabla E(\mathbf{w}_i^{(k)}))^2$.

How to Decay the Learning Rate?

An alternate approach – AdaGrad [Duchi et al 2011]

- Divide the learning rate η by the square root of the the sum of squares of gradients until that time
- This scaling factor is different for each parameter depending upon the corresponding gradients

$$\mathbf{w}_i^{(t+1)} = w^{(t)} - \frac{\eta}{\sqrt{g_i^{(t)} + \epsilon}} \nabla E(\mathbf{w}_i^{(t)})$$

where $g_i^{(t)} = \sum_{k=1}^{t} (\nabla E(\mathbf{w}_i^{(k)}))^2$.

- In a modified version AdaDelta, you take the sum of square gradients over a fixed size sliding window instead of all times from 1 to t

Momentum – Accelerating SGD Convergence

- Remember the update to w in the previous iteration, that is, $\mathbf{w}^{(t)} - \mathbf{w}^{(t-1)}$

Momentum – Accelerating SGD Convergence

- Remember the update to w in the previous iteration, that is, $\mathbf{w}^{(t)} - \mathbf{w}^{(t-1)}$
- Add it to the next iteration's update, that is,

$$w^{(t+1)} = w^{(t)} - \eta \nabla E(w^{(t)}) + \alpha(\mathbf{w}^{(t)} - \mathbf{w}^{(t-1)})$$

Momentum – Accelerating SGD Convergence

- Remember the update to w in the previous iteration, that is, $\mathbf{w}^{(t)} - \mathbf{w}^{(t-1)}$
- Add it to the next iteration's update, that is,

$$w^{(t+1)} = w^{(t)} - \eta \nabla E(w^{(t)}) + \alpha(\mathbf{w}^{(t)} - \mathbf{w}^{(t-1)})$$

- α is called the momentum, and it is typically set to around 0.9 in neural network training

Momentum – Accelerating SGD Convergence

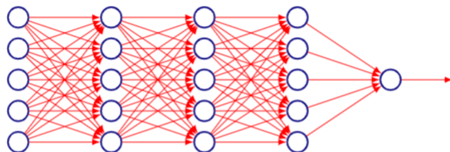
- Remember the update to w in the previous iteration, that is, $\mathbf{w}^{(t)} - \mathbf{w}^{(t-1)}$
- Add it to the next iteration's update, that is,

$$w^{(t+1)} = w^{(t)} - \eta \nabla E(w^{(t)}) + \alpha(\mathbf{w}^{(t)} - \mathbf{w}^{(t-1)})$$

- α is called the momentum, and it is typically set to around 0.9 in neural network training
- If current speed is fast, then we move even faster in the next iteration

Universality and Depth

Architecture Design



- First layer: $h^{(1)} = g^{(1)}({W^{(1)}}^T x + b^{(1)})$
- Second layer: $h^{(2)} = g^{(2)}({W^{(2)}}^T h^{(1)} + b^{(2)})$
- How do we decide *depth, width?*
- In theory how many layers *suffice?*

Universality

- Theoretical result [Cybenko, 1989]: 2-layer net with linear output with some squashing non-linearity in hidden units can approximate any continuous function over compact domain to arbitrary accuracy (given enough hidden units!)

Universality

- Theoretical result [Cybenko, 1989]: 2-layer net with linear output with some squashing non-linearity in hidden units can approximate any continuous function over compact domain to arbitrary accuracy (given enough hidden units!)
- Implication: Regardless of function we are trying to learn, a one hidden layer neural network can represent this function.

Universality

- Theoretical result [Cybenko, 1989]: 2-layer net with linear output with some squashing non-linearity in hidden units can approximate any continuous function over compact domain to arbitrary accuracy (given enough hidden units!)
- Implication: Regardless of function we are trying to learn, a one hidden layer neural network can represent this function.
- But not guaranteed that our training algorithm will be able to learn that function

Universality

- Theoretical result [Cybenko, 1989]: 2-layer net with linear output with some squashing non-linearity in hidden units can approximate any continuous function over compact domain to arbitrary accuracy (given enough hidden units!)
- Implication: Regardless of function we are trying to learn, a one hidden layer neural network can represent this function.
- But not guaranteed that our training algorithm will be able to learn that function
- Gives no guidance on how large the network will be (exponential size in worst case)

Advantages of Depth

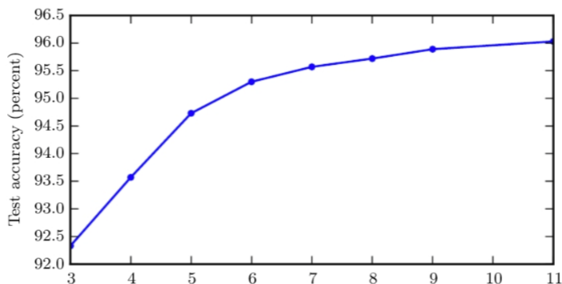


Figure 1: Goodfellow et al., 2014

- Increasing the depth of a neural network generally improves test accuracy

Advantages of Depth

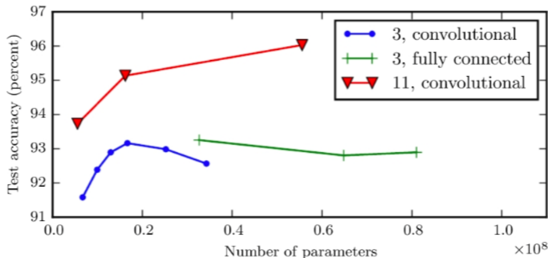


Figure 2: Goodfellow et at., 2014

- Control experiments show that other increases to model size don't yield the same effect.
- These are a lot of parameters...

Preventing Overfitting

- Approach 1 Get more data

Preventing Overfitting

- Approach 1 Get more data
  - Always best if possible!

Preventing Overfitting

- Approach 1 Get more data
  - Always best if possible!
  - If no natural ones, use data augmentation

Preventing Overfitting

- Approach 1 Get more data
  - Always best if possible!
  - If no natural ones, use data augmentation
- Approach 2 Regularization

Preventing Overfitting

- Approach 1 Get more data
  - Always best if possible!
  - If no natural ones, use data augmentation
- Approach 2 Regularization
  - Add regularizer term to the objective function

Preventing Overfitting

- Approach 1 Get more data
  - Always best if possible!
  - If no natural ones, use data augmentation
- Approach 2 Regularization
  - Add regularizer term to the objective function
  - Need to incorporate this in back-propagation

Preventing Overfitting

- Approach 1 Get more data
  - Always best if possible!
  - If no natural ones, use data augmentation
- Approach 2 Regularization
  - Add regularizer term to the objective function
  - Need to incorporate this in back-propagation
- Approach 3 Choose network structure with the right capacity:

Preventing Overfitting

- Approach 1 Get more data
  - Always best if possible!
  - If no natural ones, use data augmentation
- Approach 2 Regularization
  - Add regularizer term to the objective function
  - Need to incorporate this in back-propagation
- Approach 3 Choose network structure with the right capacity:
  - enough to fit the true regularities.

Preventing Overfitting

- Approach 1 Get more data
    - Always best if possible!
    - If no natural ones, use data augmentation
- Approach 2 Regularization
    - Add regularizer term to the objective function
    - Need to incorporate this in back-propagation
- Approach 3 Choose network structure with the right capacity:
    - enough to fit the true regularities.
    - Not enough to also fit spurious regularities (if they are weaker).

Preventing Overfitting

- Approach 1 Get more data
  - Always best if possible!
  - If no natural ones, use data augmentation
- Approach 2 Regularization
  - Add regularizer term to the objective function
  - Need to incorporate this in back-propagation
- Approach 3 Choose network structure with the right capacity:
  - enough to fit the true regularities.
  - Not enough to also fit spurious regularities (if they are weaker).
  - Requires parameter tuning, hard to guess the right size.

Preventing Overfitting

- Approach 1 Get more data
  - Always best if possible!
  - If no natural ones, use data augmentation
- Approach 2 Regularization
  - Add regularizer term to the objective function
  - Need to incorporate this in back-propagation
- Approach 3 Choose network structure with the right capacity:
  - enough to fit the true regularities.
  - Not enough to also fit spurious regularities (if they are weaker).
  - Requires parameter tuning, hard to guess the right size.
- Approach 4 Average many different models

Preventing Overfitting

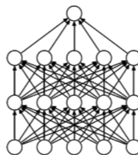
- Approach 1 Get more data
  - Always best if possible!
  - If no natural ones, use data augmentation
- Approach 2 Regularization
  - Add regularizer term to the objective function
  - Need to incorporate this in back-propagation
- Approach 3 Choose network structure with the right capacity:
  - enough to fit the true regularities.
  - Not enough to also fit spurious regularities (if they are weaker).
  - Requires parameter tuning, hard to guess the right size.
- Approach 4 Average many different models
  - Models with different forms to encourage diversity

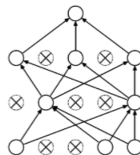
Preventing Overfitting

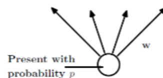
- Approach 1 Get more data
  - Always best if possible!
  - If no natural ones, use data augmentation
- Approach 2 Regularization
  - Add regularizer term to the objective function
  - Need to incorporate this in back-propagation
- Approach 3 Choose network structure with the right capacity:
  - enough to fit the true regularities.
  - Not enough to also fit spurious regularities (if they are weaker).
  - Requires parameter tuning, hard to guess the right size.
- Approach 4 Average many different models
  - Models with different forms to encourage diversity
  - Train on different subsets of data

Dropout

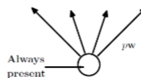
- Consider a fully connected neural net with H nodes in hidden layers.


(a) Standard Neural Net


(b) After applying dropout.



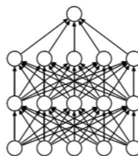
(a) At training time

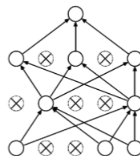


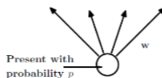
(b) At test time

Dropout

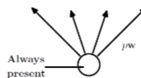
- Consider a fully connected neural net with H nodes in hidden layers.
- Each time we present a training example (for each iteration of SGD), we randomly omit each hidden unit with probability 0.5.


(a) Standard Neural Net


(b) After applying dropout.



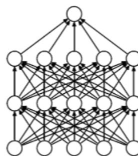
(a) At training time

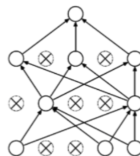


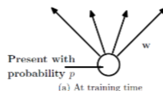
(b) At test time

Dropout

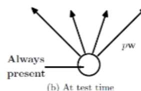
- Consider a fully connected neural net with H nodes in hidden layers.
- Each time we present a training example (for each iteration of SGD), we randomly omit each hidden unit with probability 0.5.
- So we are randomly sampling from 2^H different architectures.


(a) Standard Neural Net


(b) After applying dropout.



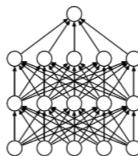
(a) At training time



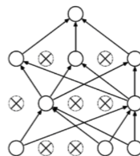
(b) At test time

Dropout

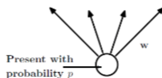
- Consider a fully connected neural net with H nodes in hidden layers.
- Each time we present a training example (for each iteration of SGD), we randomly omit each hidden unit with probability 0.5.
- So we are randomly sampling from 2^H different architectures.
- All architectures share weights.



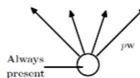
(a) Standard Neural Net



(b) After applying dropout.



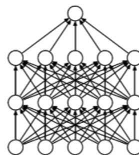
(a) At training time

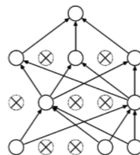


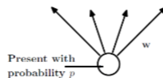
(b) At test time

Dropout as preventing co-adaptation

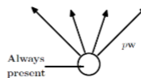
- If a hidden unit knows which other hidden units are present, it can co-adapt to them on the training data.


(a) Standard Neural Net


(b) After applying dropout.



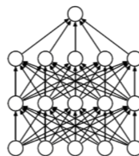
(a) At training time



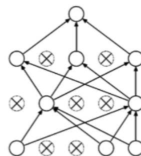
(b) At test time

Dropout as preventing co-adaptation

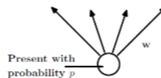
- If a hidden unit knows which other hidden units are present, it can co-adapt to them on the training data.
  - But complex co-adaptations are likely to go wrong on new test data.



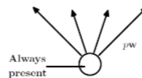
(a) Standard Neural Net



(b) After applying dropout.



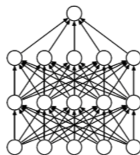
(a) At training time

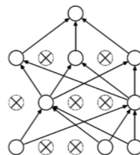


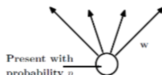
(b) At test time

Dropout as preventing co-adaptation

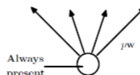
- If a hidden unit knows which other hidden units are present, it can co-adapt to them on the training data.
  - But complex co-adaptations are likely to go wrong on new test data.
  - Big, complex conspiracies are not robust.


(a) Standard Neural Net


(b) After applying dropout.



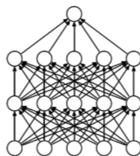
(a) At training time



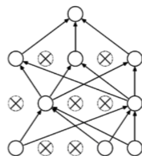
(b) At test time

Dropout as preventing co-adaptation

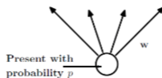
- If a hidden unit knows which other hidden units are present, it can co-adapt to them on the training data.
    - But complex co-adaptations are likely to go wrong on new test data.
    - Big, complex conspiracies are not robust.
- Dropout as orthogonalization



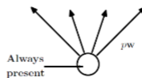
(a) Standard Neural Net



(b) After applying dropout.



(a) At training time



(b) At test time

Dropout as form of model averaging

- We sample from 2^H models. So only a few of the models ever get trained, and they only get one training example.

Dropout as form of model averaging

- We sample from 2^H models. So only a few of the models ever get trained, and they only get one training example.
- The sharing of the weights means that every model is very strongly regularized.

Dropout as form of model averaging

- We sample from 2^H models. So only a few of the models ever get trained, and they only get one training example.
- The sharing of the weights means that every model is very strongly regularized.
    - It's a much better regularizer than L2 or L1 penalties that pull the weights towards zero.
    - Note that it's hard to generalize dropout to other types of ML models, unlike L2 or L1 penalties.

What do we do at test time?

- We could sample many different architectures and take the geometric mean of their output distributions.

What do we do at test time?

- We could sample many different architectures and take the geometric mean of their output distributions.
- It better to use all of the hidden units, but to halve their outgoing weights.
  - This exactly computes the geometric mean of the predictions of all 2^H models (why?).
  - This is not exactly the same as averaging all the separate dropped out models, but it's a pretty good approximation, and it's fast.

What do we do at test time?

- We could sample many different architectures and take the geometric mean of their output distributions.
- It better to use all of the hidden units, but to halve their outgoing weights.
  - This exactly computes the geometric mean of the predictions of all 2^H models (why?).
  - This is not exactly the same as averaging all the separate dropped out models, but it's a pretty good approximation, and it's fast.
- Alternatively, run the stochastic model (i.e., the different architectures) several times on the same input.
  - This gives us an idea of the uncertainty in the answer.

Some dropout tips

- Dropout lowers your **capacity**
  - Increase network size by n/p where n is number of hidden units in original, p is probability of dropout

Some dropout tips

- Dropout lowers your **capacity**
  - Increase network size by n/p where n is number of hidden units in original, p is probability of dropout
- Dropout slows down error convergence
  - Increase learning rate by 10 to 100
  - Or increase momentum (e.g. from 0.9 to 0.99)
  - These can cause large weight growths, use weight regularization
  - May require more iterations to converge

Deep Neural Networks (DNNs)

Basic idea behind DNNs

Architecturally, a big neural network (with a lot of variants)

- in depth: 4-5 layers are commonly (Google LeNet uses more than 20)
- in width: each layer might have a few thousand hidden units
- the number of parameters: hundreds of millions, even billions

Basic idea behind DNNs

Architecturally, a big neural network (with a lot of variants)

- in depth: 4-5 layers are commonly (Google LeNet uses more than 20)
- in width: each layer might have a few thousand hidden units
- the number of parameters: hundreds of millions, even billions

Algorithmically, many new things, including:

- Pre-training: do not do error-backprogation right away
- Layer-wise greedy: train one layer at a time

Basic idea behind DNNs

Architecturally, a big neural network (with a lot of variants)

- in depth: 4-5 layers are commonly (Google LeNet uses more than 20)
- in width: each layer might have a few thousand hidden units
- the number of parameters: hundreds of millions, even billions

Algorithmically, many new things, including:

- Pre-training: do not do error-backprogation right away
- Layer-wise greedy: train one layer at a time

Computing

- Requires fast computations and coping with a lot of data
- Ex: fast Graphics Processing Unit (GPUs) are almost indispensable

Deep Convolutional Networks

- Deep supervised neural networks are generally too difficult to train

Deep Convolutional Networks

- Deep supervised neural networks are generally too difficult to train
- One notable exception: Convolutional neural networks (CNN)

Deep Convolutional Networks

- Deep supervised neural networks are generally too difficult to train
- One notable exception: Convolutional neural networks (CNN)
- Convolutional nets were inspired by the visual system's structure

Deep Convolutional Networks

- Deep supervised neural networks are generally too difficult to train
- One notable exception: Convolutional neural networks (CNN)
- Convolutional nets were inspired by the visual system's structure
- They typically have more than five layers, a number of layers which makes fully-connected neural networks almost impossible to train properly when initialized randomly.

Deep Convolutional Networks

- Compared to standard feedforward neural networks with similarly-sized layer
  - CNNs have much fewer connections and parameters
  - and so they are easier to train
  - while their theoretically-best performance is likely to be only slightly worse.

Deep Convolutional Networks

- Compared to standard feedforward neural networks with similarly-sized layer
  - CNNs have much fewer connections and parameters
  - and so they are easier to train
  - while their theoretically-best performance is likely to be only slightly worse.
- Usually applied to image datasets (where convolutions have a long history).

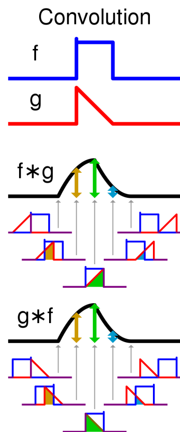
LeNet 5

Y. LeCun, L. Bottou, Y. Bengio and P. Haffner: **Gradient-Based Learning Applied to Document Recognition**, *Proceedings of the IEEE, 86(11):2278-2324, November* ***1998***

Convolution

- Continuous functions:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$
$$= \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau$$



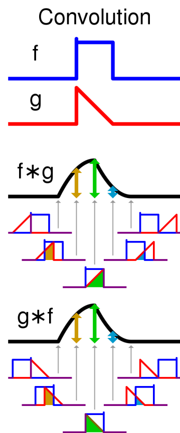
Convolution

- Continuous functions:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$
$$= \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau$$

- Discrete functions:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$
$$= \sum_{m=-\infty}^{\infty} f[n-m]g[m]$$



Convolution

- If discrete g has support on -M,...,M:

$$(f * g)[n] = \sum_{m=-M}^{M} f[n-m]g[m]$$

Where $g[m]$ is the kernel

Convolution

- If discrete g has support on -M,...,M:

$$(f * g)[n] = \sum_{m=-M}^{M} f[n-m]g[m]$$

  Where $g[m]$ is the kernel

- Product of polynomials

$$[1,2] * [3,2,5] = (x+2) * (3x^2 + 2x + 5) = 3x^3 + 8x^2 + 9x + 10$$

$$[1 \times 3 + 2 \times 0, 1 \times 2 + 2 \times 3, 1 \times 5 + 2 \times 2, 1 \times 0 + 2 \times 5] = [3, 8, 9, 10]$$

  Where $[1, 2]$ is the kernel of convolution

Convolution

- If discrete g has support on -M,...,M:

$$(f * g)[n] = \sum_{m=-M}^{M} f[n-m]g[m]$$

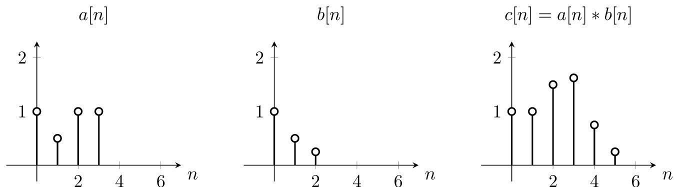
  Where $g[m]$ is the kernel

- Product of polynomials

$$[1,2] * [3,2,5] = (x+2) * (3x^2 + 2x + 5) = 3x^3 + 8x^2 + 9x + 10$$

$$[1 \times 3 + 2 \times 0, 1 \times 2 + 2 \times 3, 1 \times 5 + 2 \times 2, 1 \times 0 + 2 \times 5] = [3, 8, 9, 10]$$

  Where $[1, 2]$ is the kernel of convolution



2-Dimensional Convolution

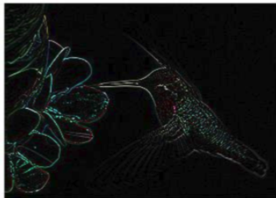
$$f[x.y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1, y-n_2]$$

https://graphics.stanford.edu/courses/cs178/applets/convolution.html

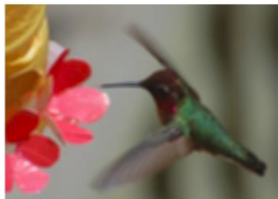
Original

Filter

0.00	0.00	0.00	0.00	0.00
0.00	0.00	-2.00	0.00	0.00
0.00	-2.00	8.00	-2.00	0.00
0.00	0.00	-2.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00



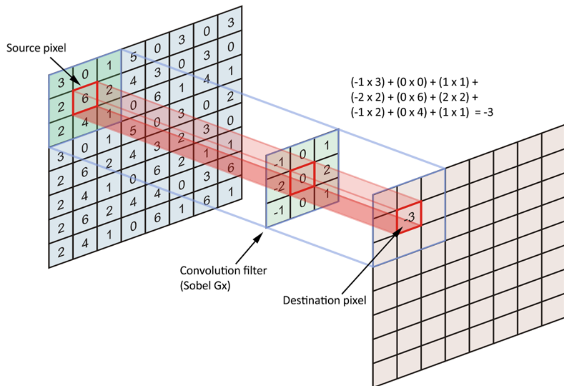
0.04	0.04	0.04	0.04	0.04
0.04	0.04	0.04	0.04	0.04
0.04	0.04	0.04	0.04	0.04
0.04	0.04	0.04	0.04	0.04
0.04	0.04	0.04	0.04	0.04



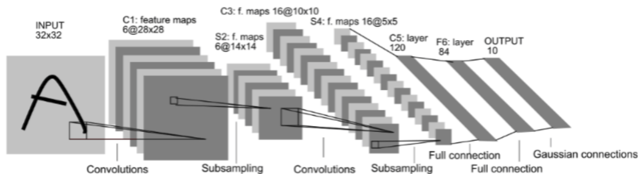
Convolutional Network Layers

Convolve subsets of an image with a small filter.

- Each pixel in the output image is a weighted sum of the filter and a subset of the input.
- Learn the values in the filter (these are your parameters, or weights).

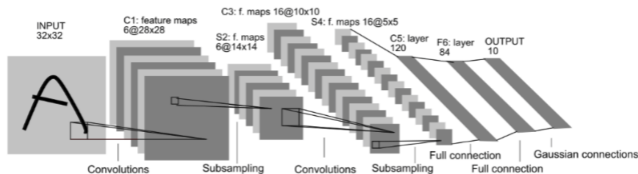


LeNet 5, LeCun 1998



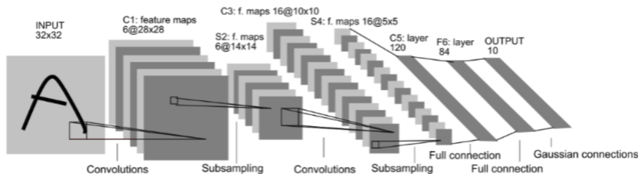
- **Input**: 32×32 pixel image. Largest character is 20×20 (All important info should be in the center of the receptive field of the highest level feature detectors)

LeNet 5, LeCun 1998



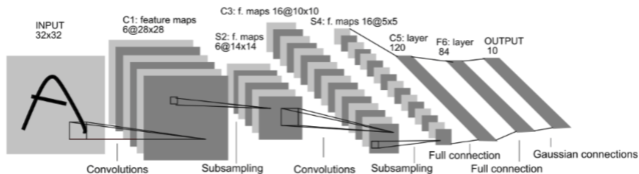
- **Input**: 32×32 pixel image. Largest character is 20×20 (All important info should be in the center of the receptive field of the highest level feature detectors)
- **Cx**: Convolutional layer (C1,C3,C5)

LeNet 5, LeCun 1998



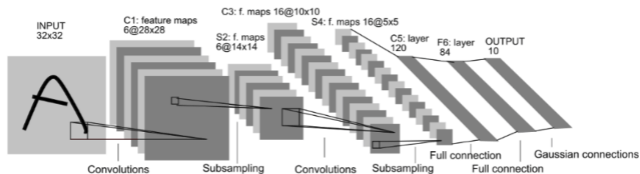
- **Input**: 32×32 pixel image. Largest character is 20×20 (All important info should be in the center of the receptive field of the highest level feature detectors)
- **Cx**: Convolutional layer (C1,C3,C5)
- **Sx**: Sub-sample layer (S2,S4)

LeNet 5, LeCun 1998



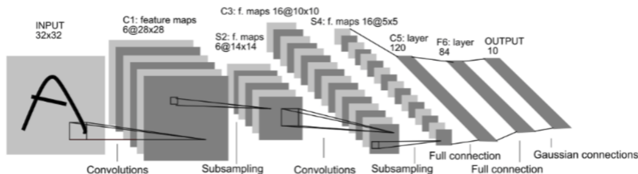
- **Input**: 32×32 pixel image. Largest character is 20×20 (All important info should be in the center of the receptive field of the highest level feature detectors)
- **Cx**: Convolutional layer (C1,C3,C5)
- **Sx**: Sub-sample layer (S2,S4)
- **Fx**: Fully connected layer (F6)

LeNet 5, LeCun 1998



- **Input**: 32×32 pixel image. Largest character is 20×20 (All important info should be in the center of the receptive field of the highest level feature detectors)
- **Cx**: Convolutional layer (C1,C3,C5)
- **Sx**: Sub-sample layer (S2,S4)
- **Fx**: Fully connected layer (F6)
- Black and White pixel values are normalized:
  Eg. White$=-0.1$, Black$=-1.175$ (Mean of pixels $= 0$, Standard deviation of pixels $= 1$)

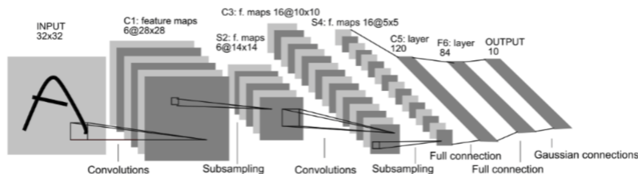
LeNet 5, Layer C1



C1: Convolutional layer with 6 feature maps of size 28X28 $C1^k (k = 1..6)$

Each unit of C1 has 5x5 receptive field in the input layer.

LeNet 5, Layer C1

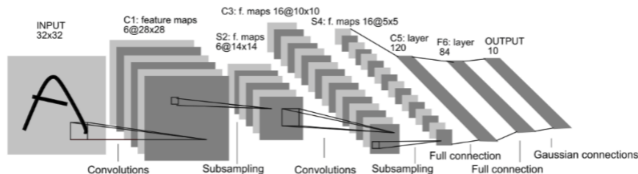


C1: Convolutional layer with 6 feature maps of size 28X28 $C1^k(k = 1..6)$
Each unit of C1 has 5x5 receptive field in the input layer.

- Topological structure
- Sparse connections
- Shared weights

$(5 * 5 + 1) * 6 = 156$ parameters to learn
Connections: $28 * 28 * (5 * 5 + 1) * 6 = 122304$
If it was fully connected, we had (32*32+1)*(28*28)*6 parameters

LeNet 5, Layer S2



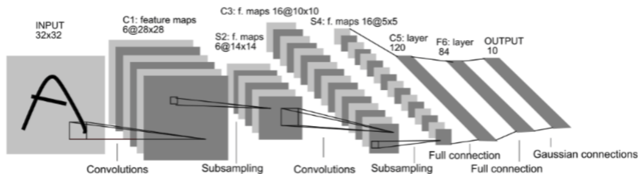
S2: Sub-sampling layer with 6 feature maps of size 14×14

2×2 non-overlapping receptive fields in C1

$$S2_{ij}^k = tanh(w_1^k \sum_{s,t=0}^{1} C1_{2i-s,2j-t}^k + w_2^k)$$

Layer S2: 6*2=12 trainable parameters

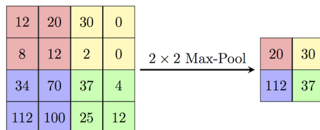
Connections: $14 * 14 * (2 * 2 + 1) * 6 = 5880$



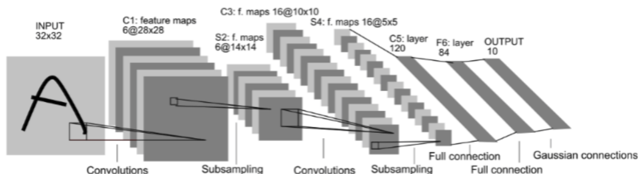
S2: Sub-sampling layer with 6 feature maps of size 14×14

2×2 non-overlapping receptive fields in C1

These days, we typically use

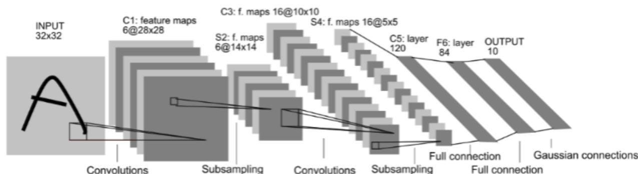


LeNet 5, Layer C5



- C5: Convolutional layer with 120 feature maps of size 1x1
- Each unit in C5 is connected to all 16 5x5 receptive fields in S4

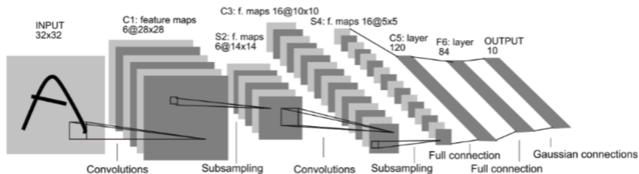
LeNet 5, Layer C5



- C5: Convolutional layer with 120 feature maps of size 1x1
- Each unit in C5 is connected to all 16 5x5 receptive fields in S4

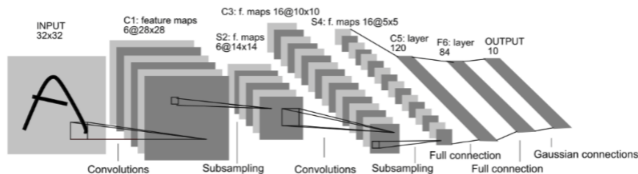
Layer C5: $120 * (16 * 25 + 1) = 48120$ trainable parameters and connections (Fully Connected)

LeNet 5, Layer F6



- Layer F6: 84 fully connected nodes. 84*(120+1)=10164 trainable parameters and connections.

LeNet 5, Layer F6

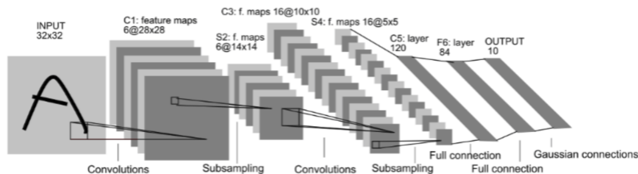


- Layer F6: 84 fully connected nodes. 84*(120+1)=10164 trainable parameters and connections.
- Output layer: 10 RBF (One for each digit)

$$y_i = \sum_{j=1}^{84} (x_j - w_{ij})^2$$

Where i =1,2,..10

LeNet 5, Layer F6



- Layer F6: 84 fully connected nodes. 84*(120+1)=10164 trainable parameters and connections.
- Output layer: 10 RBF (One for each digit)

$$y_i = \sum_{j=1}^{84} (x_j - w_{ij})^2$$

Where i =1,2,..10

$$84 = 7x12, \text{ stylized image}$$

Weight update: Backpropagation

GoogLeNet (Szegedy et al., 2015)

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Today's networks can go much deeper than LeNet!

Recurrent Neural Networks

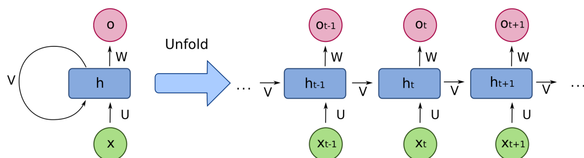
- Used to model temporal data (e.g., speech recognition).

Recurrent Neural Networks

- Used to model temporal data (e.g., speech recognition).
- Results can flow backwards (we use hidden node outputs from previous times as inputs to the current node).

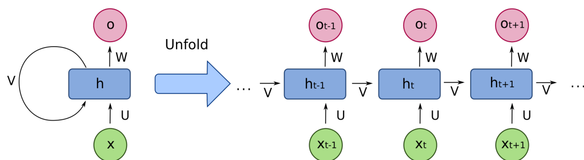
Recurrent Neural Networks

- Used to model temporal data (e.g., speech recognition).
- Results can flow backwards (we use hidden node outputs from previous times as inputs to the current node).
- Creates an "internal state" of the hidden node input/outputs.



Recurrent Neural Networks

- Used to model temporal data (e.g., speech recognition).
- Results can flow backwards (we use hidden node outputs from previous times as inputs to the current node).
- Creates an "internal state" of the hidden node input/outputs.



- Several variants, e.g., long short-term memory (LSTM) networks.

Summary

You should know:

- Advantages of depth in neural networks.
- How to use dropout to prevent overfitting.
- Differences between a convolutional and feedforward neural network.