18-661 Introduction to Machine Learning

Neural Networks-II

Spring 2025

ECE – Carnegie Mellon University

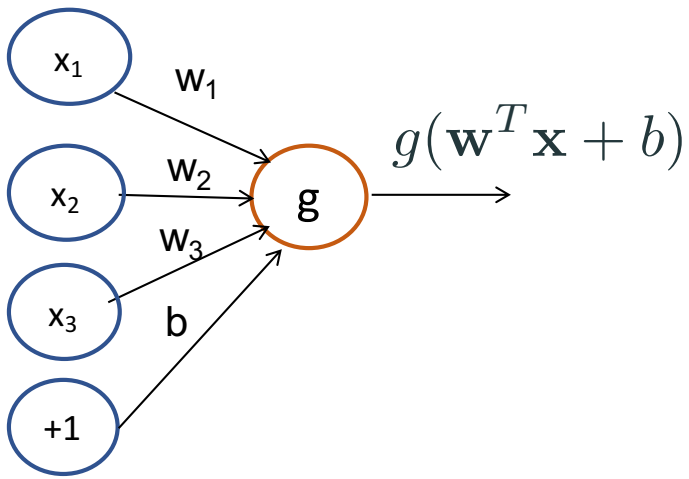
Outline

1. Review: Multi-layer Neural Network
2. Review: Choosing Neural Network Architectures
3. Training a Neural Network: Backpropagation
4. Optimizing SGD Parameters for Faster Convergence
5. Mitigating Overfitting

Artificial Neuron Model

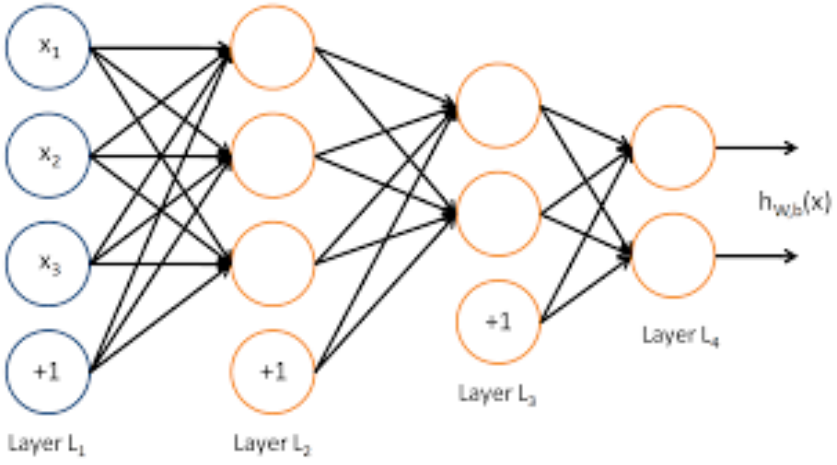
Based on the biological insights, a mathematical model for an 'artificial' neuron was developed

- Each input x_i is multiplied by weight w_i
- Add a $+1$ input neuron which is multiplied by the bias b
- Apply a non-linear function g to the weighted combination of the inputs, $\mathbf{w}^T\mathbf{x} + b$
- Different candidates for g: heaviside function, sigmoid, tanh, rectified linear unit, etc.



Single Artificial Neuron

Neural Networks Learn Hierarchical Features



- Start with feature vector **x** containing all pixels in the image
- Layer 1: distill the edges of the image
- Layer 2: distill triangles, circles, etc.
- Layer 3: recognize pointy ears, fur style etc.
- Layer 4: performs logistic regression on the features in layer 3

We cannot directly control what each layer learns; this depends on the training data

Learned Hierarchical Representations

Learned representations using CNN trained on ImageNet:

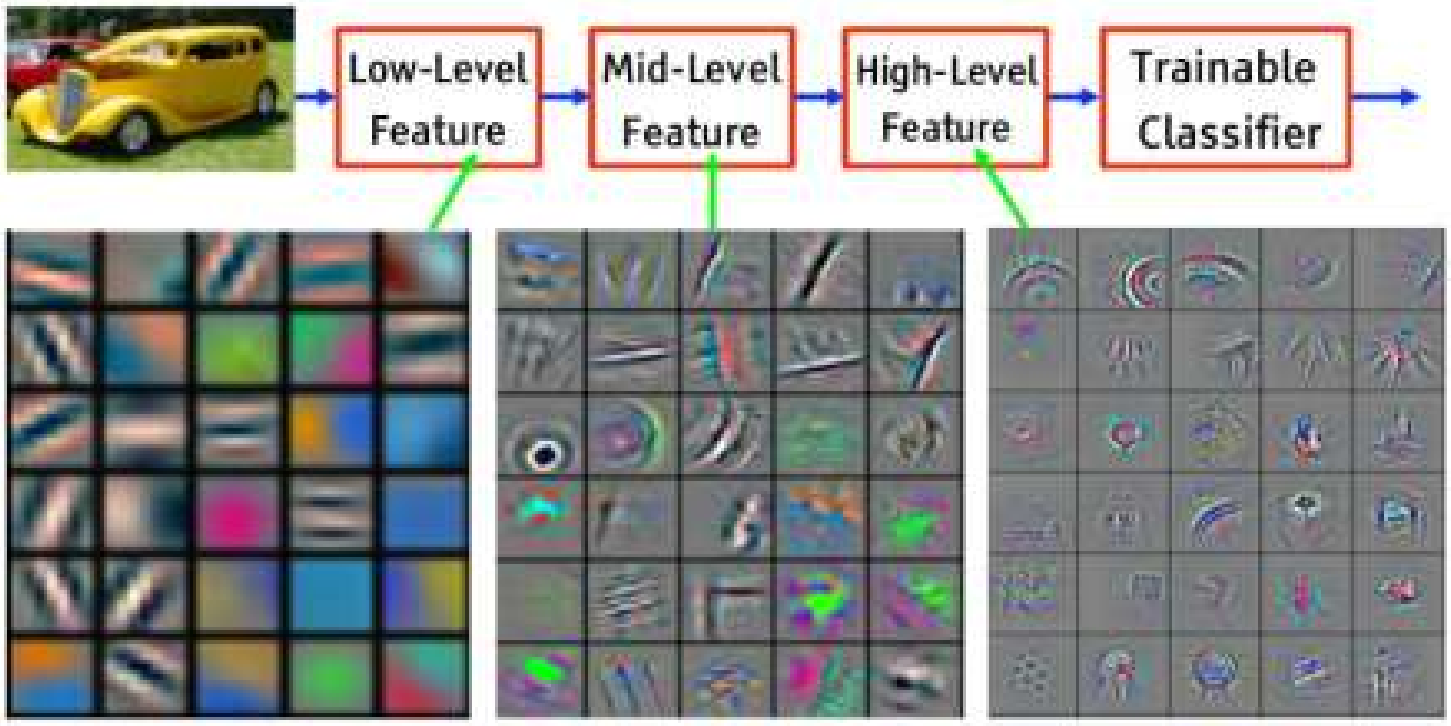
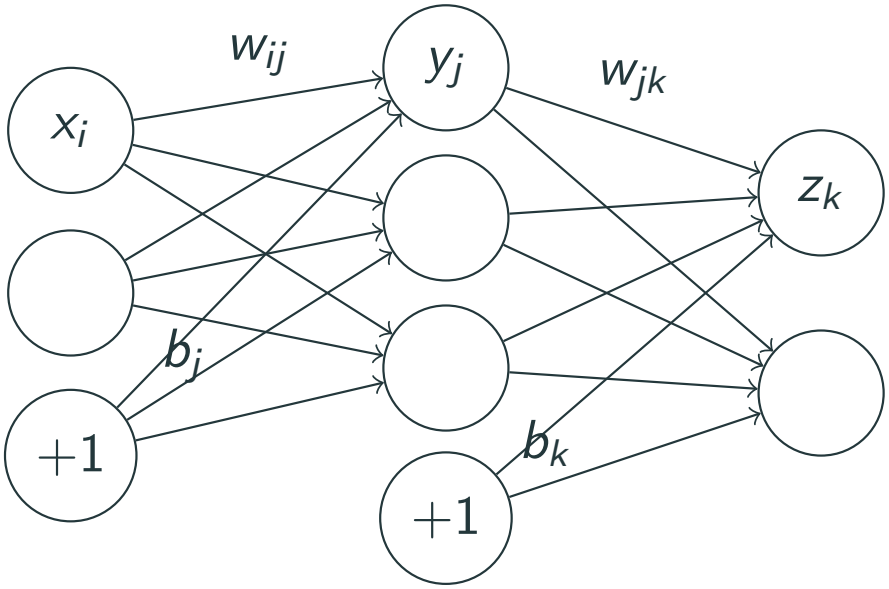
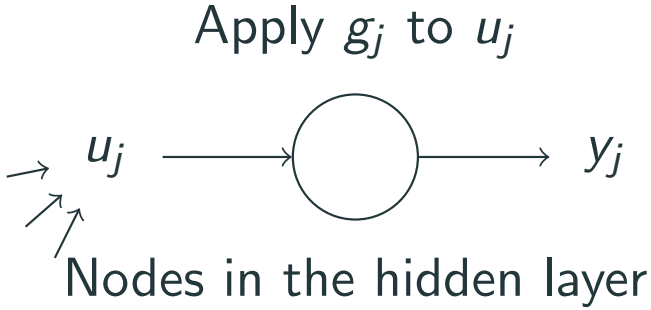


Figure credit: Y. Lecun's slide with research credit to, Zeiler and Fergus, 2013.

Review: Multi-layer Neural Network

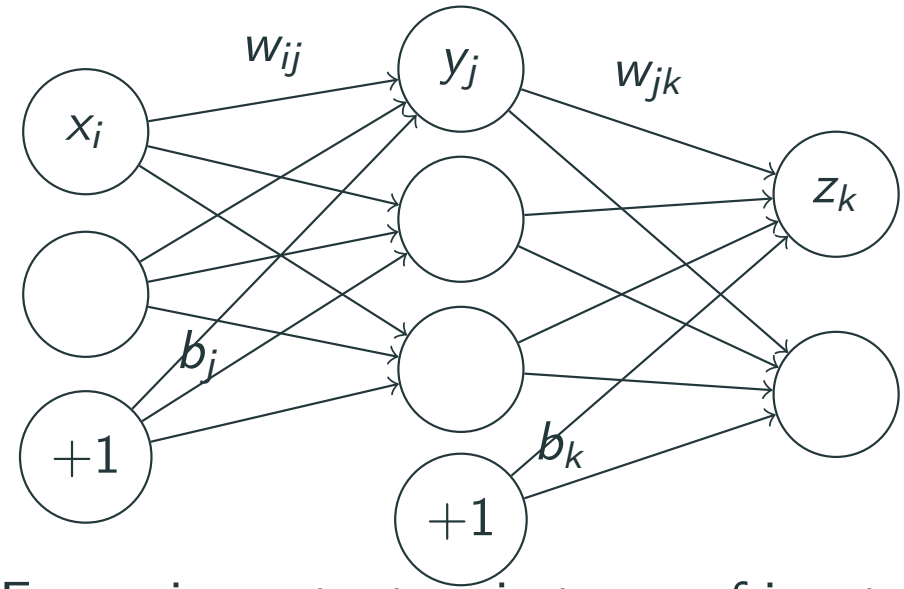
Multi-layer Neural Network



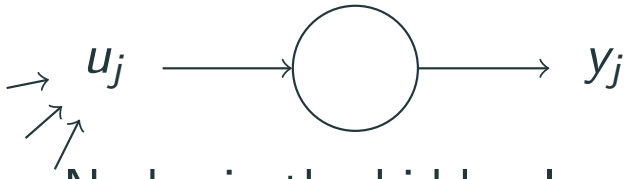


- w_{ij}: weights connecting node i in layer $(\ell - 1)$ to node j in layer ℓ.
- b_j, b_k: bias for nodes in layers j and k respectively.
- u_j, u_k: inputs to nodes j and k (where $u_j = b_j + \sum_i x_i w_{ij}$).
- g_j, g_k: activation function for node j (applied to u_j) and node k.
- $y_j = g_j(u_j)$,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

Forward Propagation in Neural Networks



Apply g_j to u_j



Nodes in the hidden layer

Expressing outputs z in terms of inputs x is called forward-propagation.

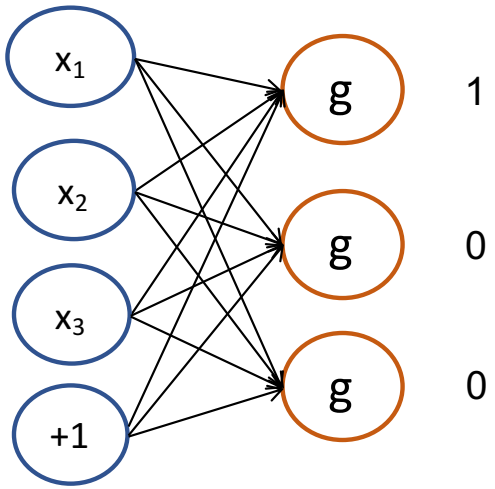
- Express inputs u_j to the hidden layer in terms of x:
  $u_j = \sum_i w_{ij} x_i + b_j$
- Express outputs y_j of the hidden layer in terms of x:
  $y_j = g(\sum_i w_{ij} x_i + b_j)$
- Express inputs u_k to the final layer in terms of x
- Express outputs z_k of the final layer in terms of x:
  $z_k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

Softmax for Multi-class Classification

- If the target is takes C possible values
- If y belongs to the first class, the outputs should be $[1, 0, \ldots, 0]$
- Need to produce a vector $\hat{y}$ with $\hat{y}_i = \mathbb{P}\mathrm{r}(y = i|x)$

Linear output layer ($g(x) = x$) first produces un-normalized log probabilities:

$$z = \mathbf{w}^T \mathbf{x} + b, \quad \mathrm{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

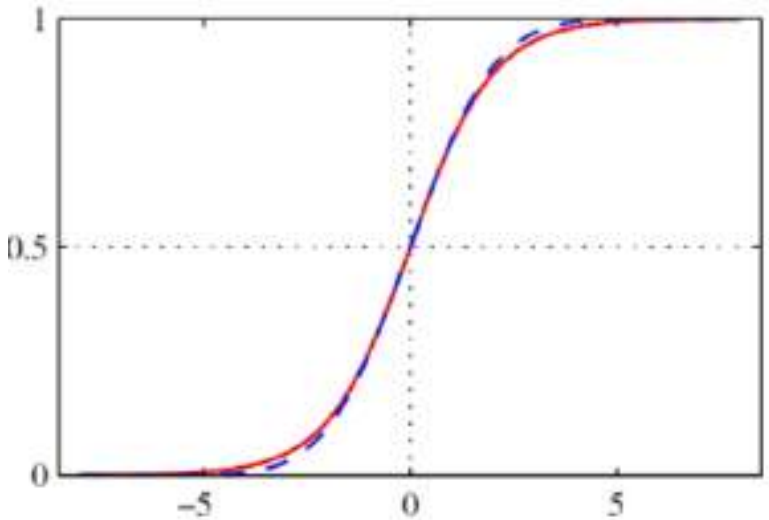


Multiclass Regression for $C = 3$

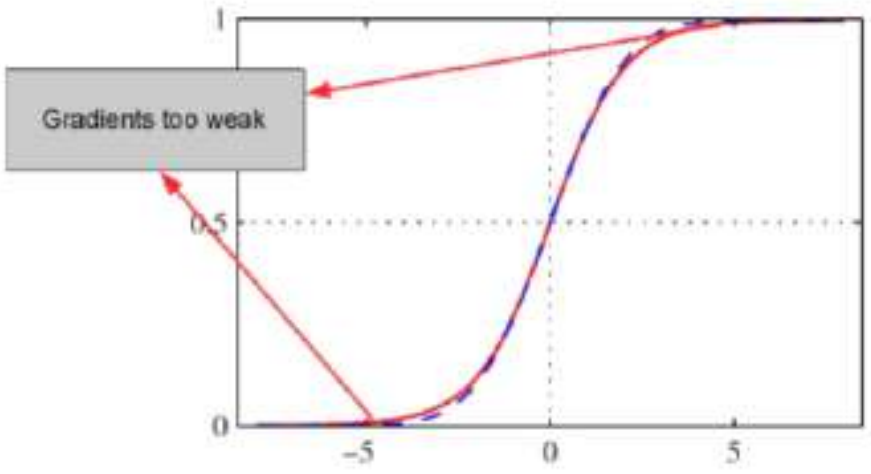
Sigmoid Activation Function

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

- Squashing type non-linearity: pushes output to range [0,1]

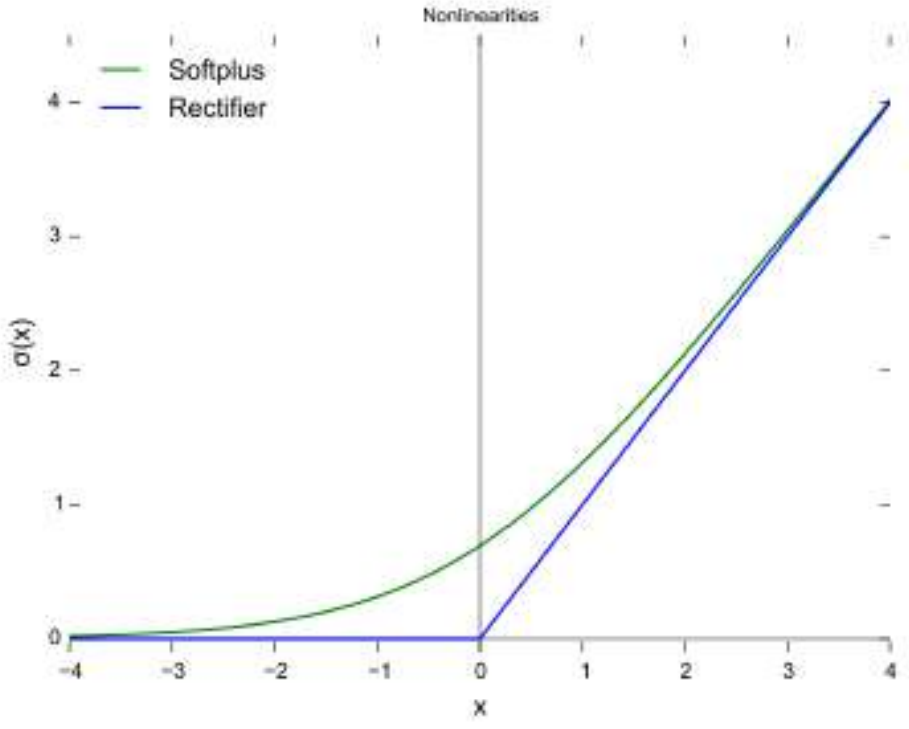


The Vanishing Gradients Problem



- Problem: Near-constant value across most of their domain, strongly sensitive only when z is closer to zero
- Saturation makes gradient-based learning difficult and inefficient

Rectified Linear Units



- Approximates the softplus function which is $\log(1 + e^z)$
- ReLu Activation function is $g(z) = max(0, z)$ with $z \in R$
- Similar to linear units. Easy to optimize!
- Give large and *consistent* gradients when active

Activation Choices for Each Layer

Output layer produces a classification decision.

- Probabilities of the input being in each class.
- Often uses sigmoid, softmax, or tanh activations.

Hidden layers convert activated inputs to classification features.

- ReLU, ELU and variants are popular choices.
- Highly problem dependent!

Input layer initially transforms the features.

Often uses linear, sigmoid, or tanh activations.

Review: Choosing Neural Network Architectures

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.
- In practice, increasing the depth of a neural network generally improves test accuracy.

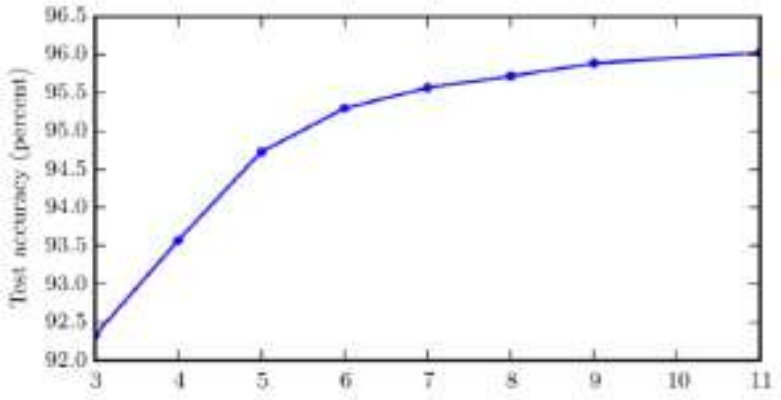
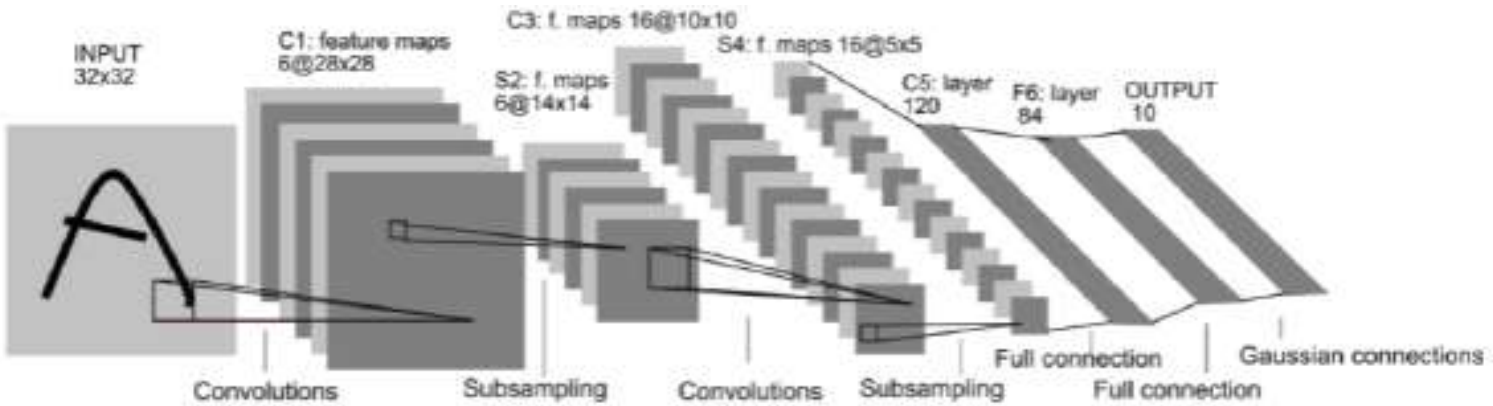


Figure 2: Goodfellow et al., 2014

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.



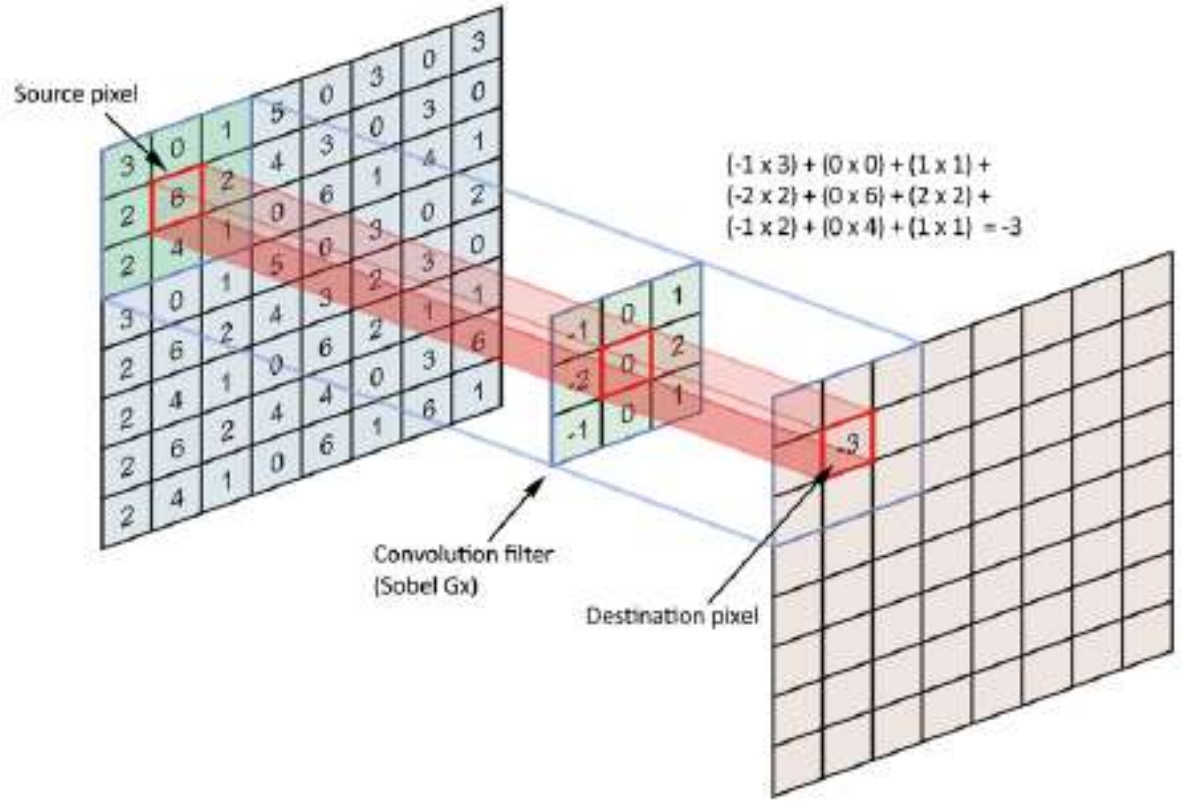
Example: LeNet 5 (LeCun, 1998).

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).

Many fewer parameters (and connections) than a feedforward network.



Training a Neural Network: Backpropagation

Learning Parameters

How to learn the parameters?

- Choose the right loss function
  - Regression: Least-square loss (today's class)

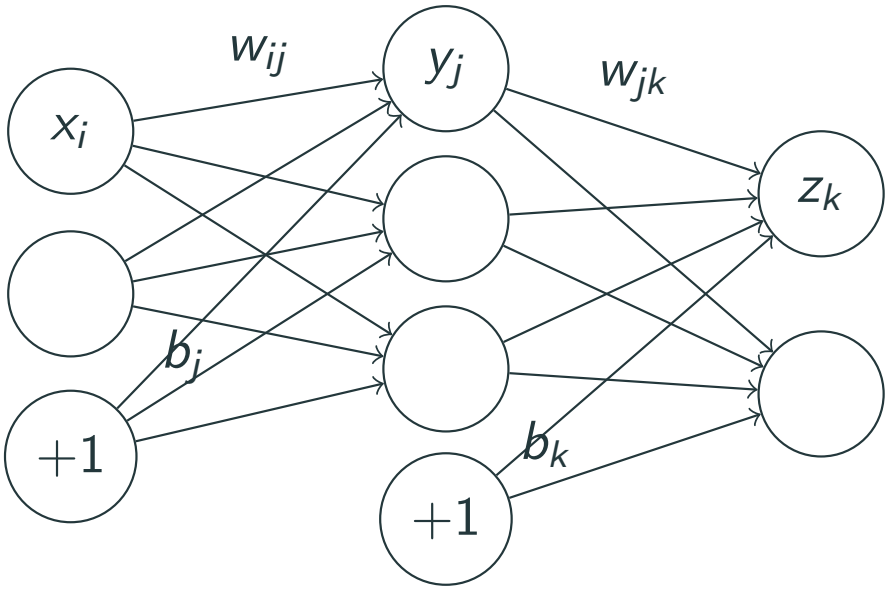
$$min \sum_n (f(\mathbf{x}_n) - t_n)^2$$

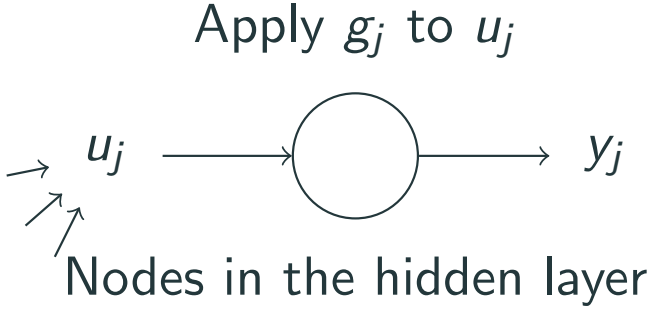
  - Classification: cross-entropy loss (in the homework)

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

- Hard optimization problem because f (the output of the neural network) is (usually) a complicated function of $\mathbf{x}_n$
  - Stochastic gradient descent is commonly used
  - Many optimization tricks are applied

Stochastic Gradient Descent





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

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

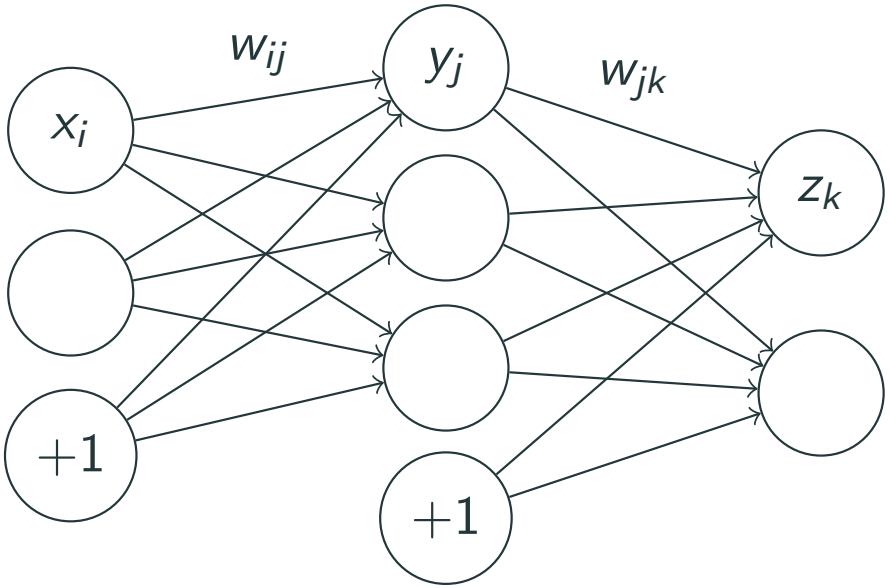
- Update the parameters: $\mathbf{w} \leftarrow \mathbf{w} - \eta\Delta$
- Iterate the process until some (pre-specified) stopping criteria

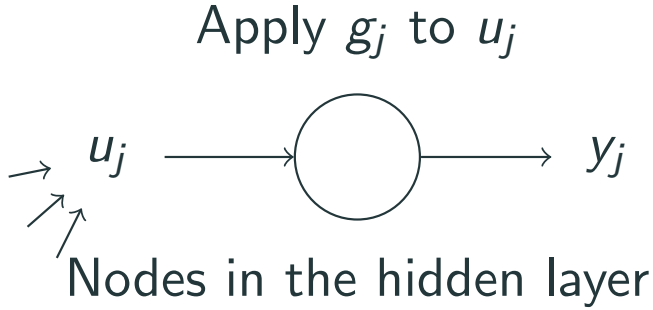
Updating the Parameter Values

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

- Step 1: Forward-propagate to find the output 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.

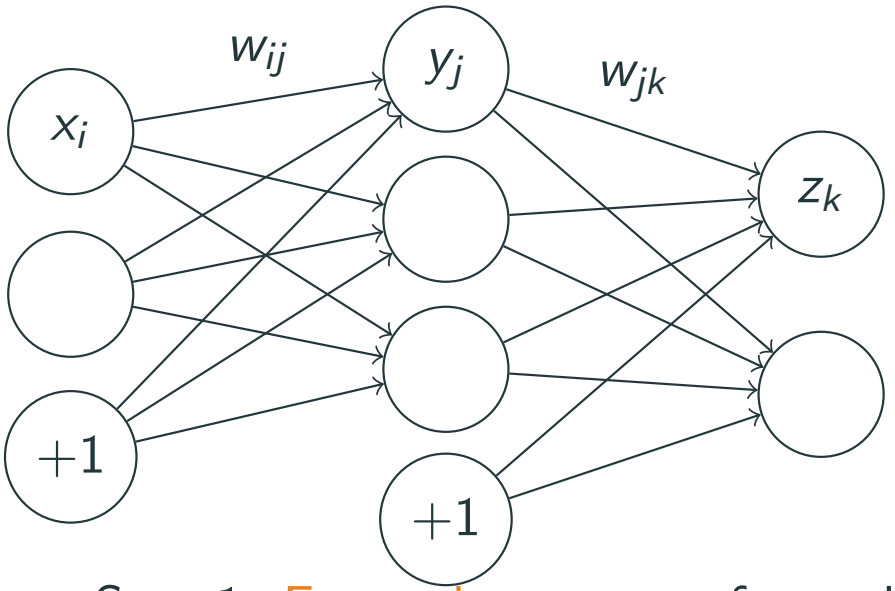
Illustrative Example

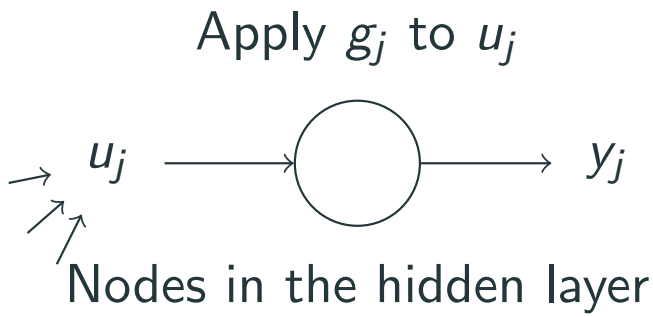




- w_{ij}: weights connecting node i in layer $(\ell - 1)$ to node j in layer ℓ.
- b_j, b_k: bias for nodes j and k.
- u_j, u_k: inputs to nodes j and k (where $u_j = b_j + \sum_i x_i w_{ij}$).
- g_j, g_k: activation function for node j (applied to u_j) and node k.
- $y_j = g_j(u_j)$, $z_k = g_k(u_k)$: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

Illustrative Example (Steps 1 and 2)



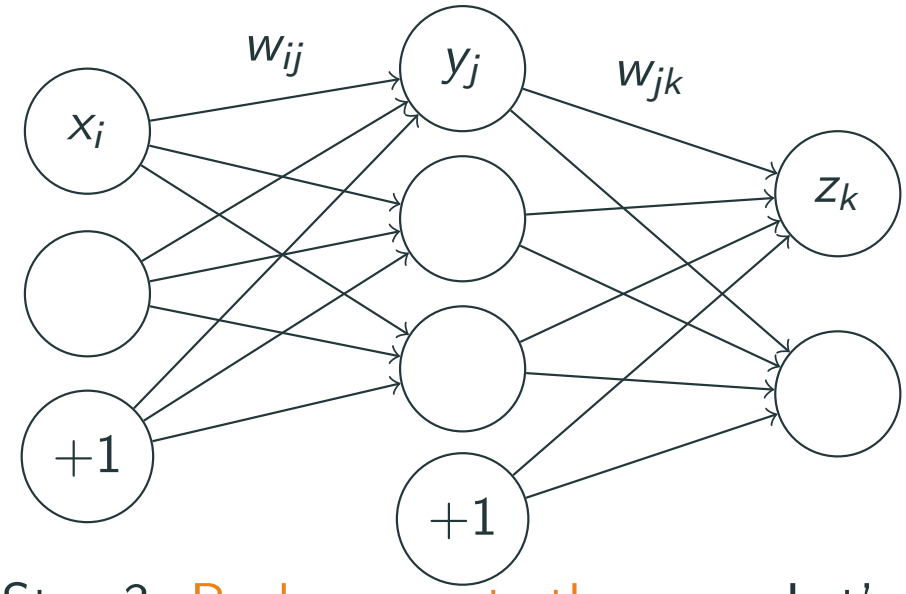


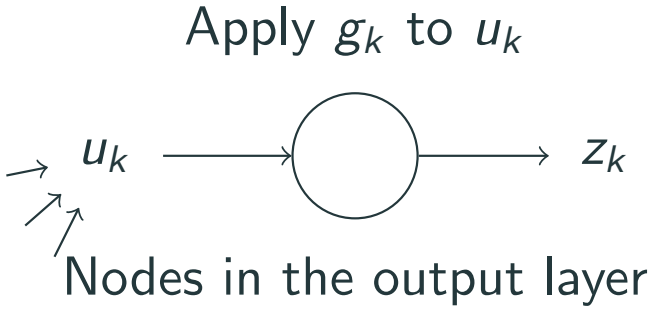
- Step 1: Forward-propagate for each output z_k.

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

- 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)





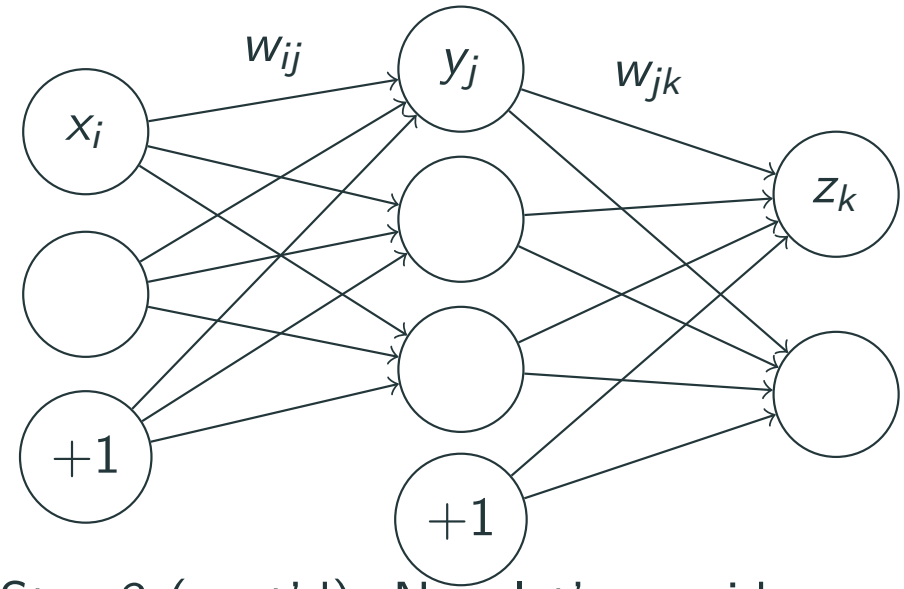
Step 3: Backpropagate the error. Let's start at the output layer with weight w_{jk}, recalling that $E = \frac{1}{2}\sum_{k\in K}(z_k - t_k)^2$, $u_k = b_k + \sum_j w_{jk}y_j$:

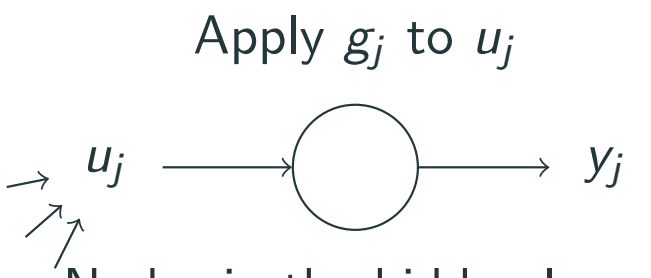
$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial u_k}\frac{\partial u_k}{\partial w_{jk}} = (z_k - t_k)\frac{\partial z_k}{\partial u_k}\frac{\partial u_k}{\partial w_{jk}}$$

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

where $\delta_k = (z_k - t_k)g_k'(u_k) = \frac{\partial E}{\partial u_k}$ is called the error in u_k.

Illustrative Example (Step 3, Hidden Layer)





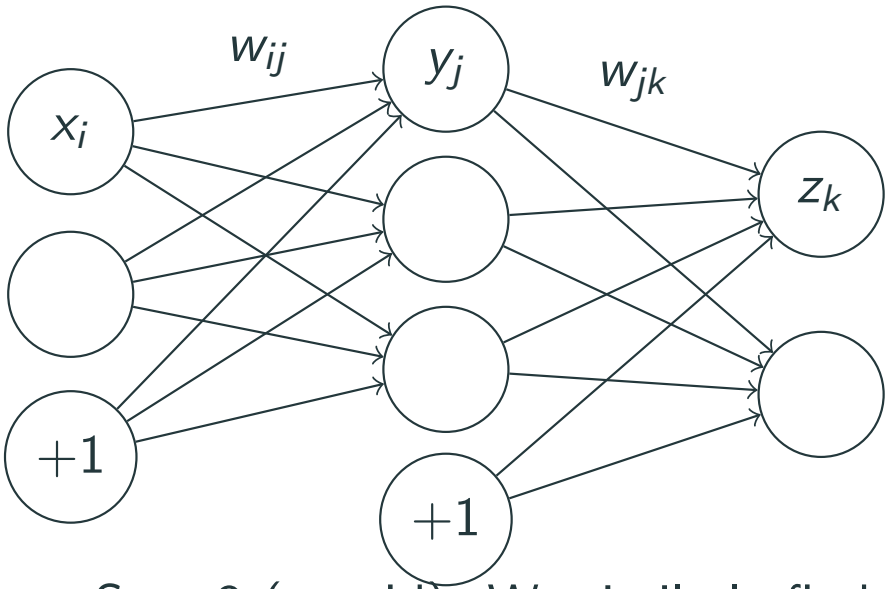
Step 3 (cont'd): Now let's consider w_{ij} in the hidden layer, recalling $u_j = b_i + \sum_i x_i w_{ij}$, $u_k = b_k + \sum_j g_j(u_j) w_{jk}$, $z_k = g_k(u_k)$:

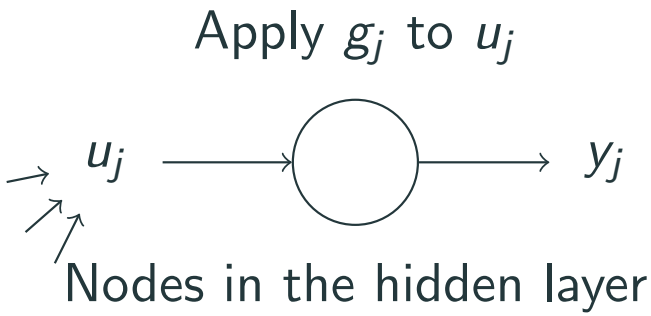
$$\frac{\partial E}{\partial w_{ij}} = \sum_{k \in K} \frac{\partial E}{\partial u_k} \frac{\partial u_k}{\partial y_j} \frac{\partial y_j}{\partial u_j} \frac{\partial u_j}{\partial w_{ij}} = \sum_{k \in K} \delta_k w_{jk} \frac{\partial y_j}{\partial u_j} \frac{\partial u_j}{\partial w_{ij}}$$

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

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

Illustrative Example (Steps 3 and 4)



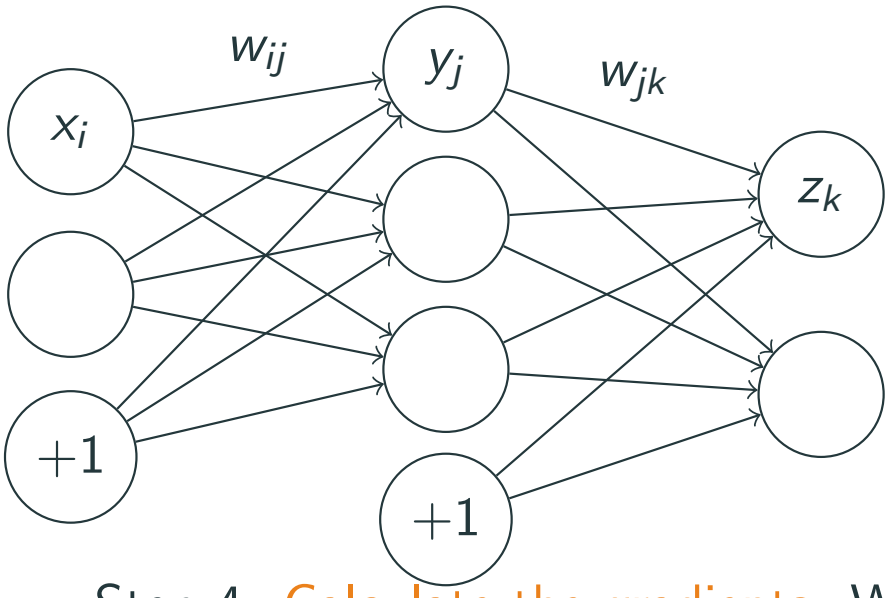


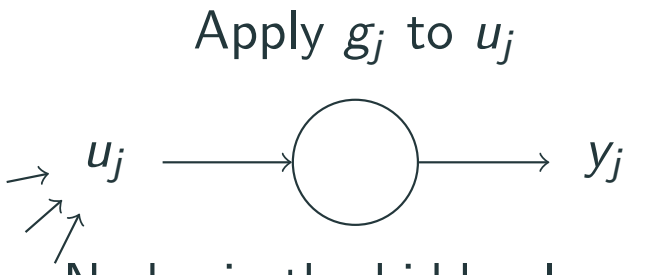
- Step 3 (cont'd): We similarly find that $\frac{\partial E}{\partial b_k} = \delta_k$, $\frac{\partial E}{\partial b_j} = \delta_j$.
- Step 4: Calculate the gradients. We have found that

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

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)





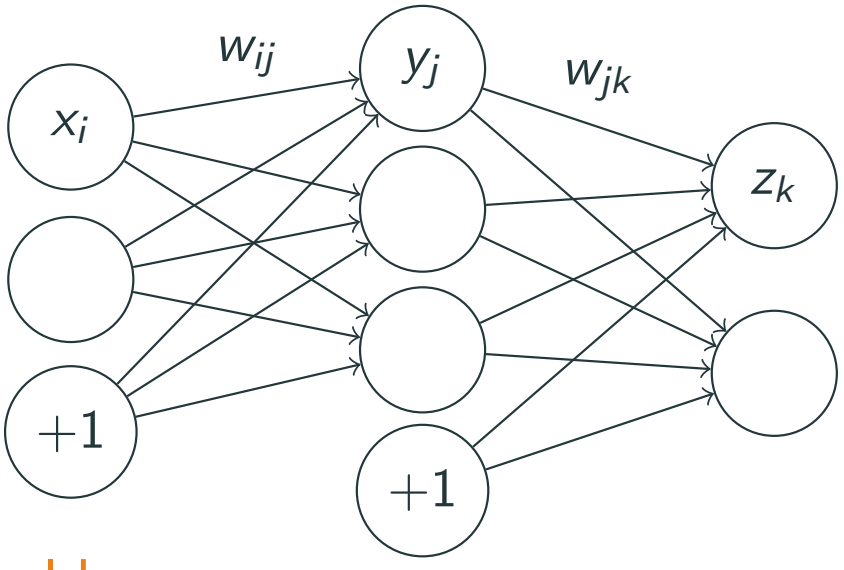
- Step 4: Calculate the gradients. We have found that

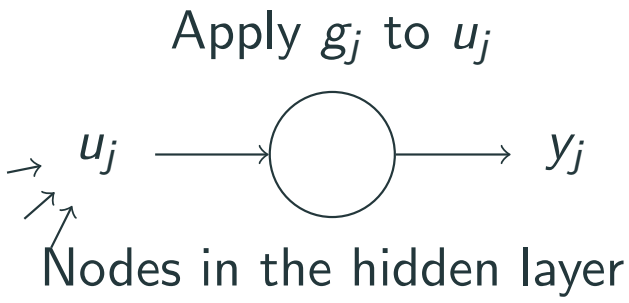
$$\frac{\partial E}{\partial w_{ij}} = \delta_j x_i \text{ and } \frac{\partial E}{\partial w_{jk}} = \delta_k y_j.$$

  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

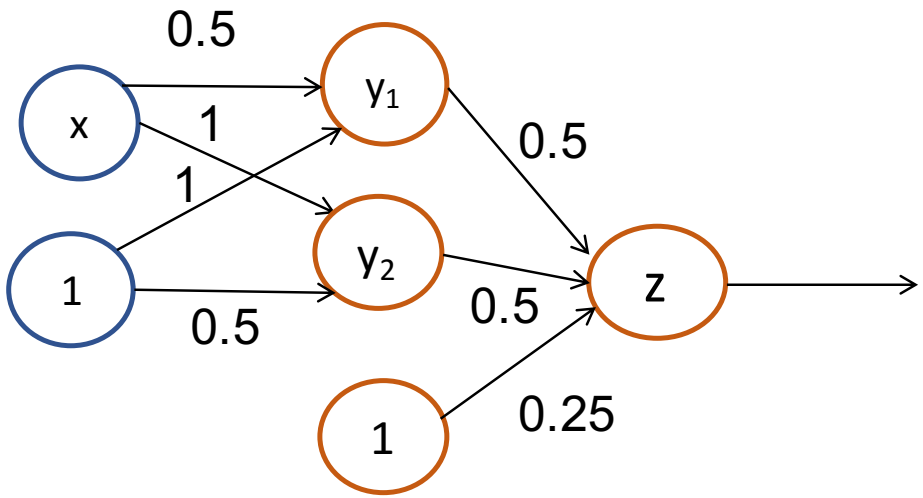
- Error in each of its outputs is $\frac{\partial E}{\partial z_k} = z_k - t_k$.
- Error in input u_k to the final layer is $\delta_k = \frac{\partial E}{\partial u_k} = g_k'(u_k)(z_k - t_k)$

Hidden Layer

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

The gradients w.r.t. w_{jk} and w_{ij} are $y_j \delta_k$ and $x_i \delta_j$.

Exercise: Back-Propagation

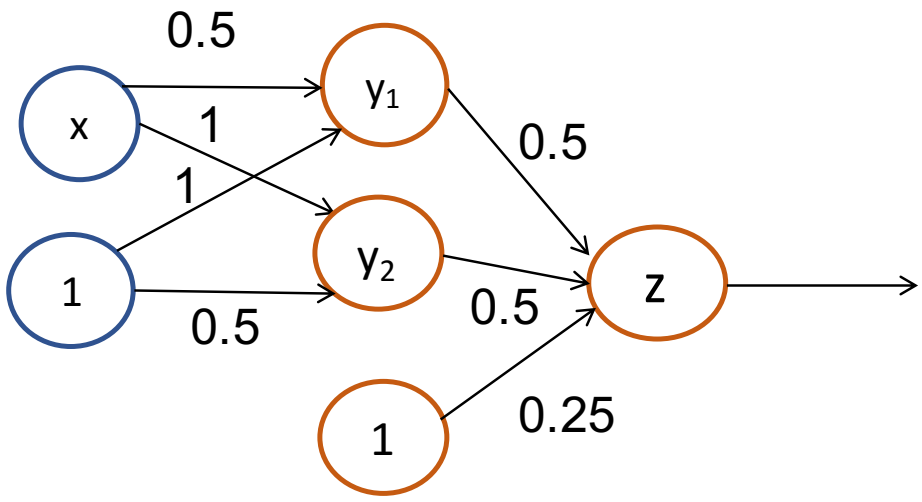


Suppose the output $z = 0.9$ but the target is 1. Perform backpropagation and compute the gradient of error w.r.t. the weight connecting x and y_2.

Forward-propagation

- $y_1 = \sigma(0.5x + 1)$ and $y_2 = \sigma(x + 0.5)$
- Input to last layer $u = 0.5y_1 + 0.5y_2 + 0.25$
- Final Output $z = \sigma(u)$

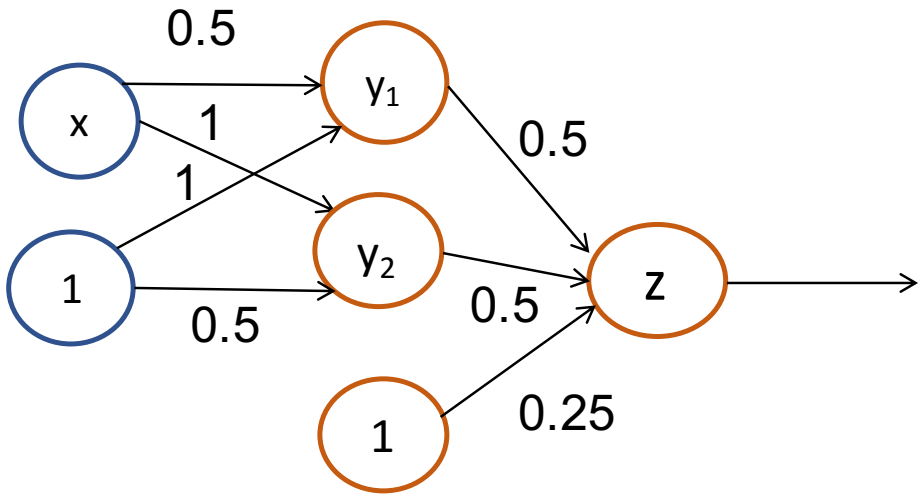
Exercise: Back-Propagation



Final layer

- Error in output z is $0.9 - 1 = -0.1$
- Error in input u is $-0.1 \times \sigma'(u)$

Exercise: Back-Propagation

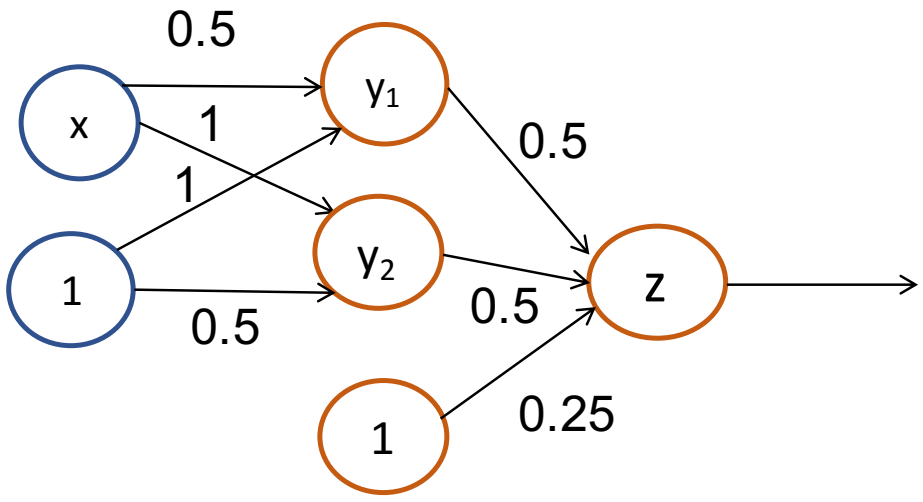


Hidden Layer

- Error in y_1 is $-0.1 \times \sigma'(u) \times 0.5$
- Error in y_2 is $-0.1 \times \sigma'(u) \times 0.5$
- Error in u_2 is $-0.1 \times \sigma'(u) \times 0.5 \times \sigma'(u_2)$

The gradient w.r.t. the weight connecting x and y_2 is $-0.1 \times \sigma'(u) \times 0.5 \times \sigma'(u_2) \times x$

Exercise: Back-Propagation



Hidden Layer

- The gradient w.r.t. the weight connecting x and y_2 is

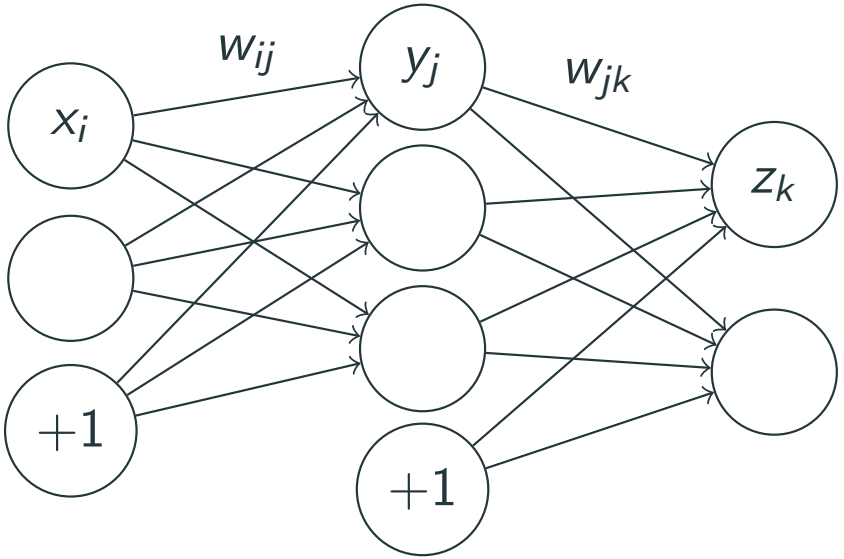
$$\frac{\partial E}{\partial w} = -0.1 \times \sigma'(u) \times 0.5 \times \sigma'(u_2) \times x$$

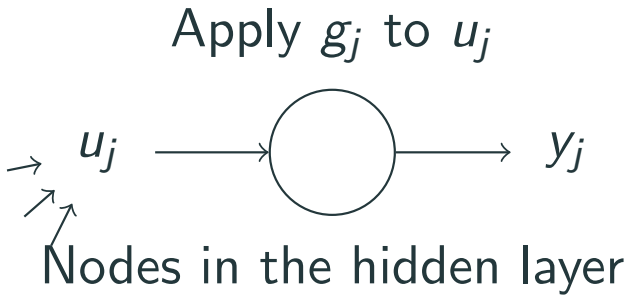
- Thus, we will update the weight as

$$w \leftarrow w - \eta \frac{\partial E}{\partial w}$$

Vectorized Implementation

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



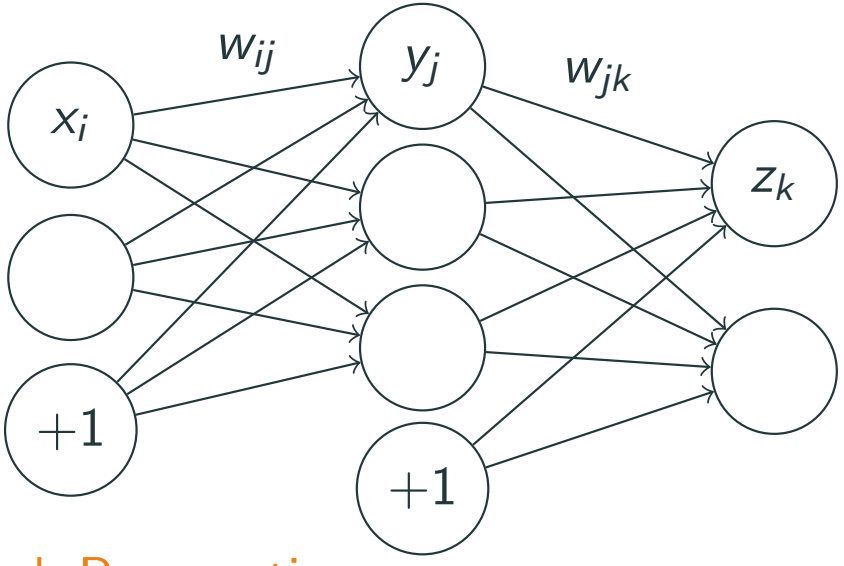


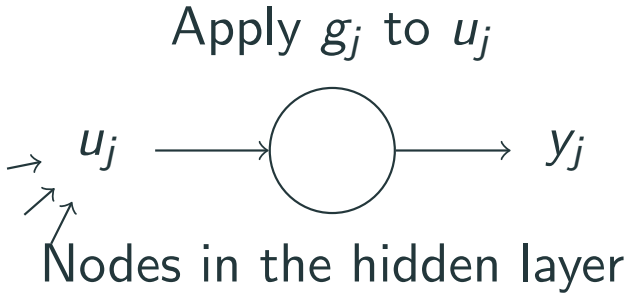
Forward-Propagation

- Represent the weights between layers $\ell - 1$ and ℓ as a matrix $\mathbf{W}^{(\ell)}$, where the (j, k) entry in $\mathbf{W}^{(\ell)}$ is w_{jk}.
- Outputs of layer $\ell - 1$ are in a row vector $\mathbf{y}^{(\ell-1)}$. Then we have $\mathbf{u}^{(\ell)} = \mathbf{y}^{(\ell-1)}\mathbf{W}^{(\ell)}$.
- Outputs of layer ℓ are in the row vector $\mathbf{y}^{(\ell)} = g(\mathbf{u}^{(\ell)})$.

Vectorized Implementation

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





Back-Propagation

- For each layer l find $\Delta^{(\ell)}$, the vector of errors in $\mathbf{u}^{(\ell)}$ in terms of the final error E.
- Update weights $\mathbf{W}^{(\ell)}$ using $\Delta^{(\ell)}$
- Recursively find $\Delta^{(\ell-1)}$ in terms $\Delta^{(\ell)}$:
  $\Delta^{(\ell-1)} = g'\left(\mathbf{u}^{(\ell-1)}\right) \cdot \left(\mathbf{W}^{(\ell)}\Delta^{(\ell)}\right)^T$, where $\cdot$ denotes element-wise multiplication.

Optimizing SGD Parameters for Faster Convergence

(Full-)Batch vs. Stochastic GD

- 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 layer, the batch gradients are

$$\frac{\partial E}{\partial w_{jk}} = \sum_{n=1}^{N}(z_{k,n} - t_{k,n})$$

- 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
- 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

How to Choose Mini-Batch Size

- 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

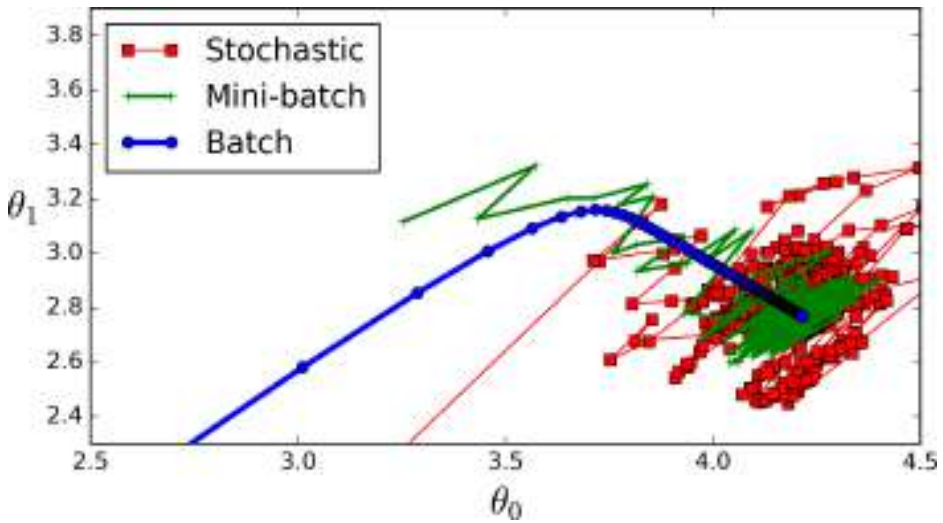


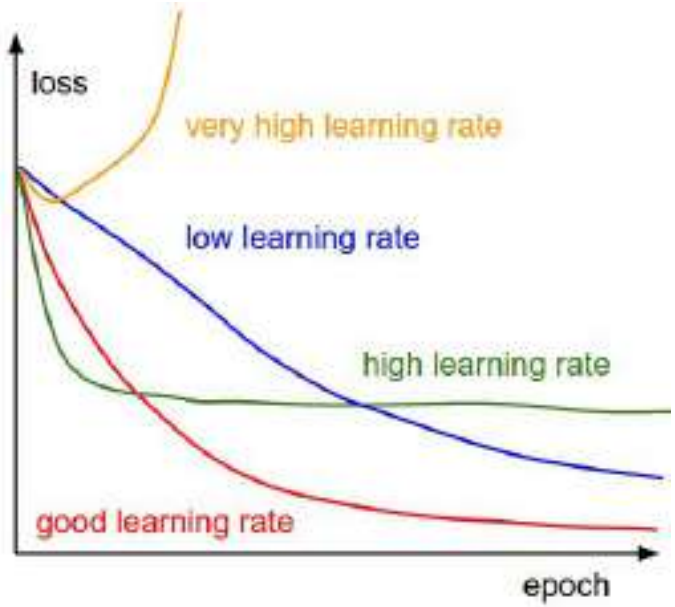
Image source: `https://github.com/buomsoo-kim/Machine-learning-toolkits-with-python`

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 $\eta = 0.1$
- 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

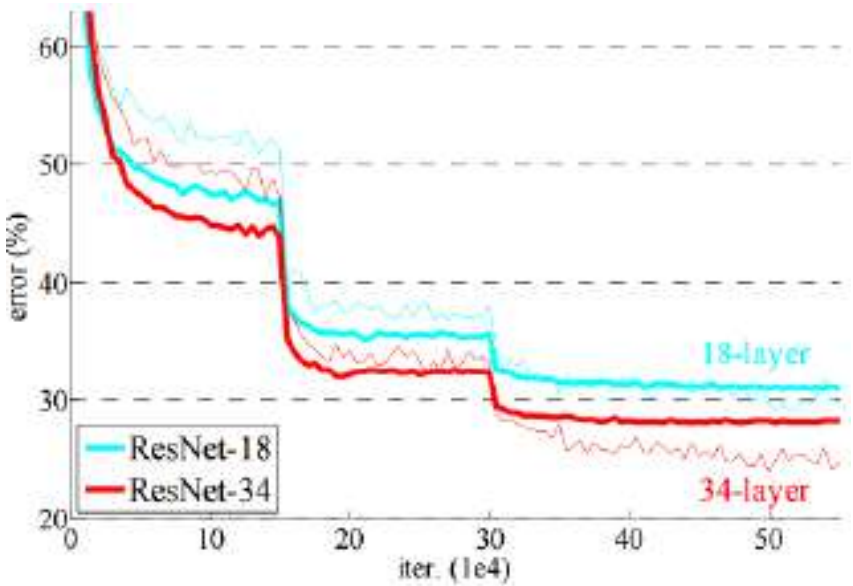


Image Source: `http://www.bdhammel.com/learning-rates/`

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
- 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

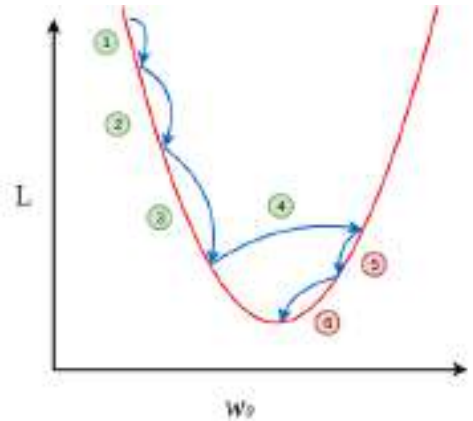
Momentum

- Drawback of vanilla SGD: Makes steady but often slow progress towards the minimum, analogous to moving at a constant velocity

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

- When $\nabla E(w^{(t)})$ is small, SGD slows down

- Momentum SGD: Considers the inertia of motion to speed-up future steps – if the gradient $\nabla E(w^{(t)})$ in step $t+1$ is large, a fraction of it also gets added to the $t+2$, $t+3$, .. $t+n$-th updates.
- Analogous to rolling a heavy ball down a slope gaining speed.

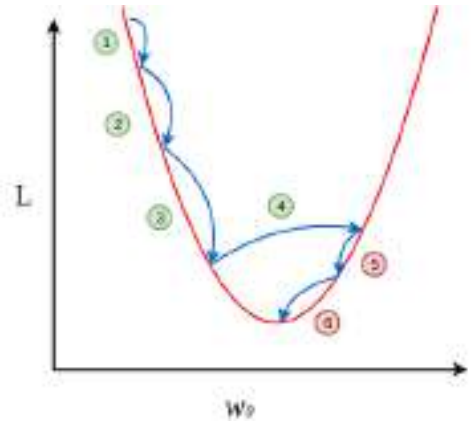


Momentum

- Momentum SGD: Accelerates SGD by considering the inertia of motion to speed-up future steps.
- Formal update rule is given by:

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

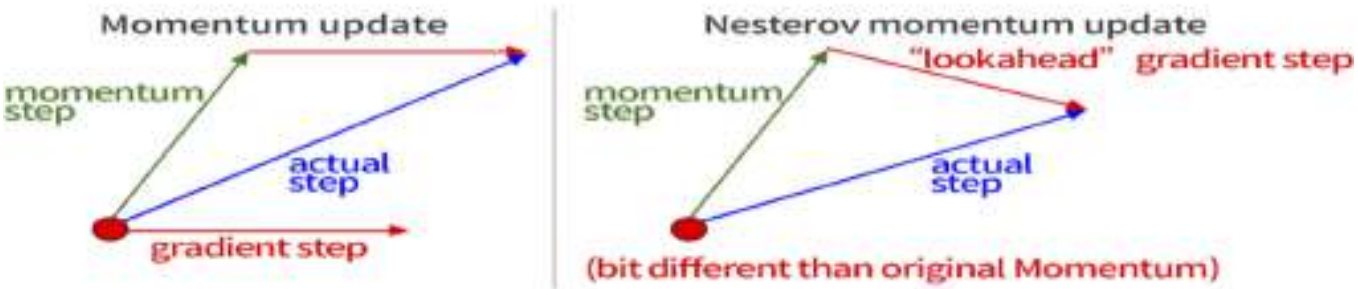
  where the update vector $v^{(t)}$ in the $t+1$-th iteration includes the gradient plus a small fraction γ of the previous update vector
- The momentum γ is usually set to 0.9



Nesterov Momentum

- Momentum can speed-up the ball too much and cause it to overshoot the minimum
- Nesterov momentum modifies the update rule slightly so as to anticipate future changes in the gradient

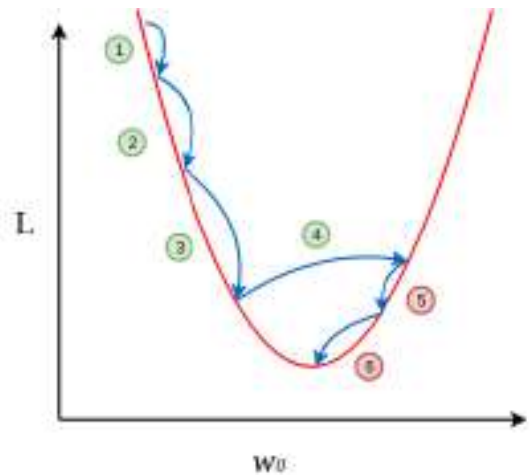
$$v^{(t)} = \gamma v^{(t-1)} + \eta \nabla E(w^{(t)} - \gamma v^{(t-1)})$$
$$w^{(t+1)} = w^{(t)} - v^{(t)}$$



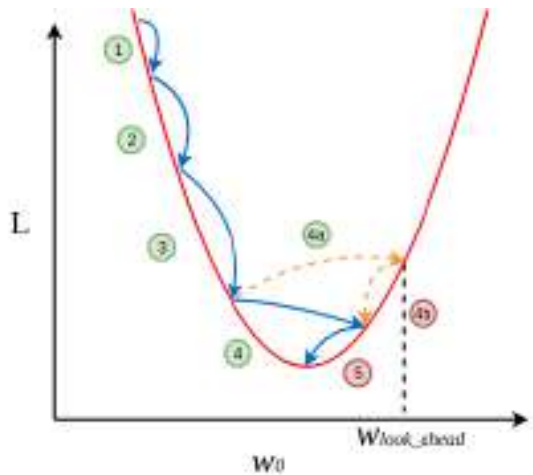
Nesterov Momentum

- Momentum can speed-up the ball too much and cause it to overshoot the minimum
- Nesterov momentum modifies the update rule slightly so as to anticipate future changes in the gradient

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



(a) Momentum-Based Gradient Descent



(b) Nesterov Accelerated Gradient Descent

$\Longrightarrow \frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Positive(+)}$ $\Longrightarrow \frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Negative(-)}$

Adam

- Adaptive Moment Estimation (Adam) is a method that first appeared in [Kingma, Ba 2015] – it combines the adaptive learning rate idea used in AdaDelta with an adaptive momentum term

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \quad \text{First moment gradient estimate} \quad (1)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \quad \text{Second moment gradient estimate} \quad (2)$$

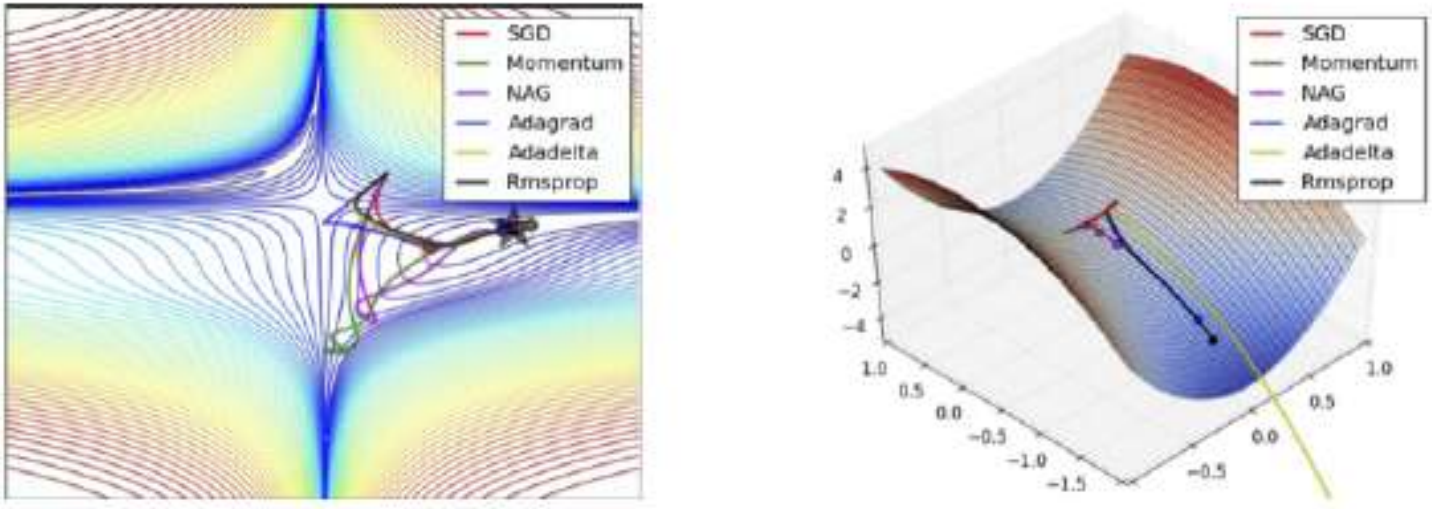
- Model parameters are updated as:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \left(\frac{\eta}{\sqrt{v_t/(1-\beta_2^t) + \epsilon}} \right) \frac{m_t}{1-\beta_1^t}$$

- Other adaptive methods include RMSprop, AdaMax, Nadam etc.

Which optimizer should one use?

- For sparse parameter vectors, adaptive optimizers work very well, and Adam typically outperforms other adaptive methods. However, the exact trend varies with the dataset and loss function



(a) SGD optimization on loss surface contours

(b) SGD optimization on saddle point

Figure 4: Source and full animations: Alec Radford

Full animation can be found here: `http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html`

Mitigating Overfitting

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 (Dropout)
  - Train on different subsets of data

Summary

You should know:

- How to train a neural network using the back-propagation algorithm.
- Effect of learning rate and mini-batch size on training speed and accuracy
- Optimization methods such as AdaGrad and Adam
- Some general strategies to prevent overfitting.