

Business Analytics & Machine Learning

Neural Networks

Prof. Dr. Martin Bichler

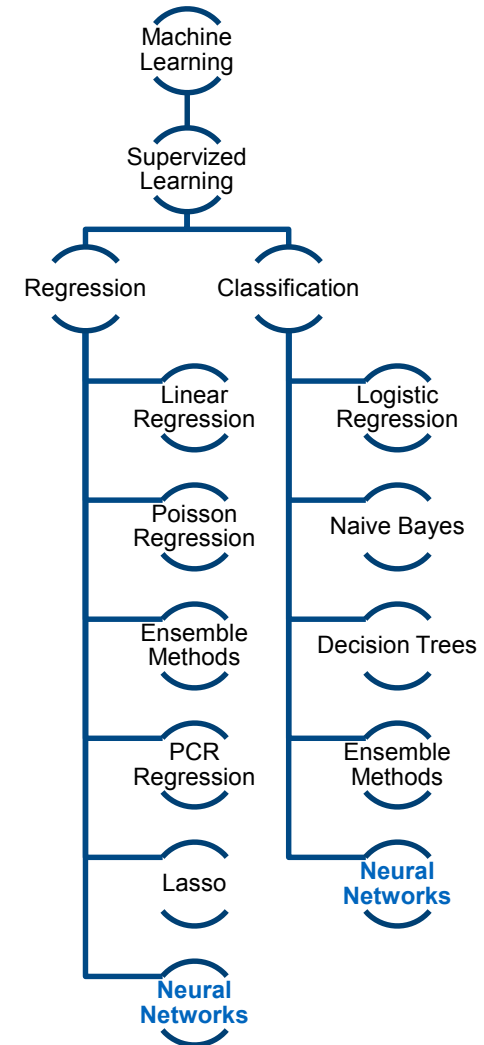
Department of Computer Science

School of Computation, Information, and Technology

Technical University of Munich

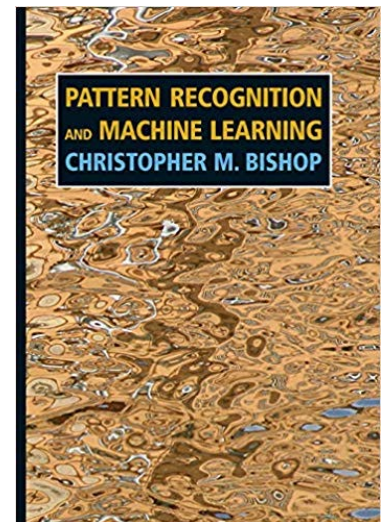
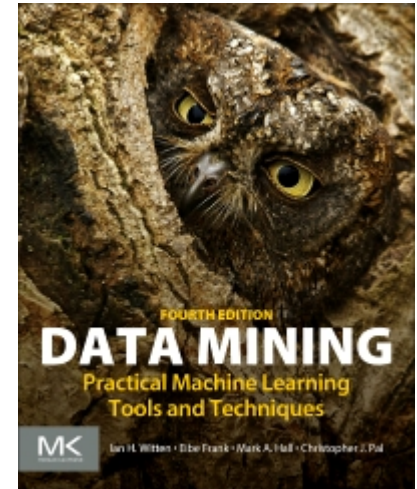
Course Content

- Introduction
- Regression Analysis
- Regression Diagnostics
- Logistic and Poisson Regression
- Naive Bayes and Bayesian Networks
- Decision Tree Classifiers
- Data Preparation and Causal Inference
- Model Selection and Learning Theory
- Ensemble Methods and Clustering
- Dimensionality Reduction
- Convex Optimization
- **Neural Networks**
- Reinforcement Learning



Recommended Literature

- **Data Mining: Practical Machine Learning Tools and Techniques**
 - Ian H. Witten, Eibe Frank, Mark Hall, Christopher Pal
 - <http://www.cs.waikato.ac.nz/ml/weka/book.html>
 - Section 7, 10
- **Pattern Recognition and Machine Learning**
 - Christopher M. Bishop
 - Section 5



Agenda for Today

Empirical risk minimization in neural networks

Backpropagation

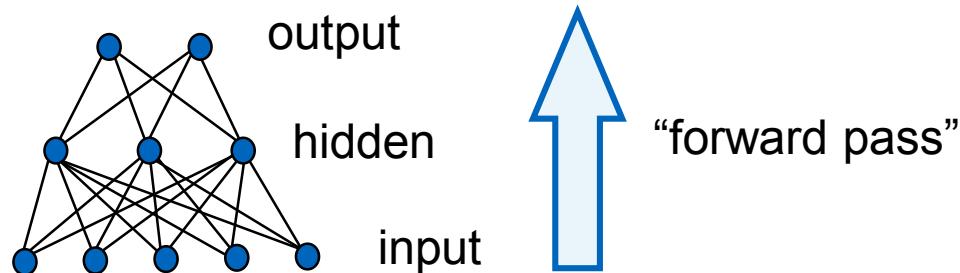
Deep learning



Multi-Layer Feed-Forward Networks

Multi-layer networks can represent arbitrary non-linear functions.

A typical multi-layer network consists of an input, hidden and output layer, each fully connected to the next, with activation feeding forward.

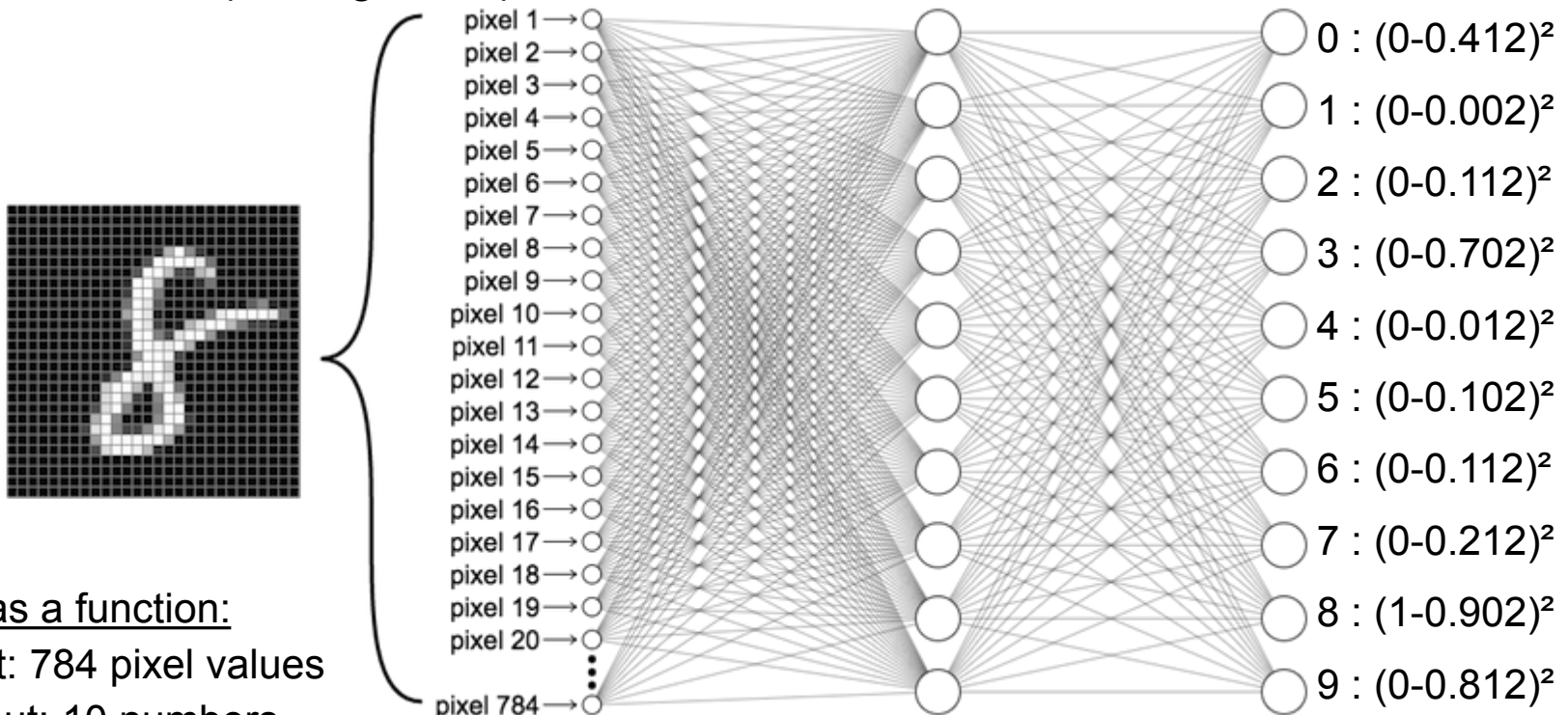


Multi-layer networks have thousands of weights and biases that need to be adapted. The weights determine the overall function computed.

Weights are updated via backpropagation (i.e. gradient decent).

Loss Function

Use a loss function (aka. cost function) to compute the average (misclassification) cost for all training data, i.e. for a particular set of weights and biases it computes a single number (average cost).



NN as a function:

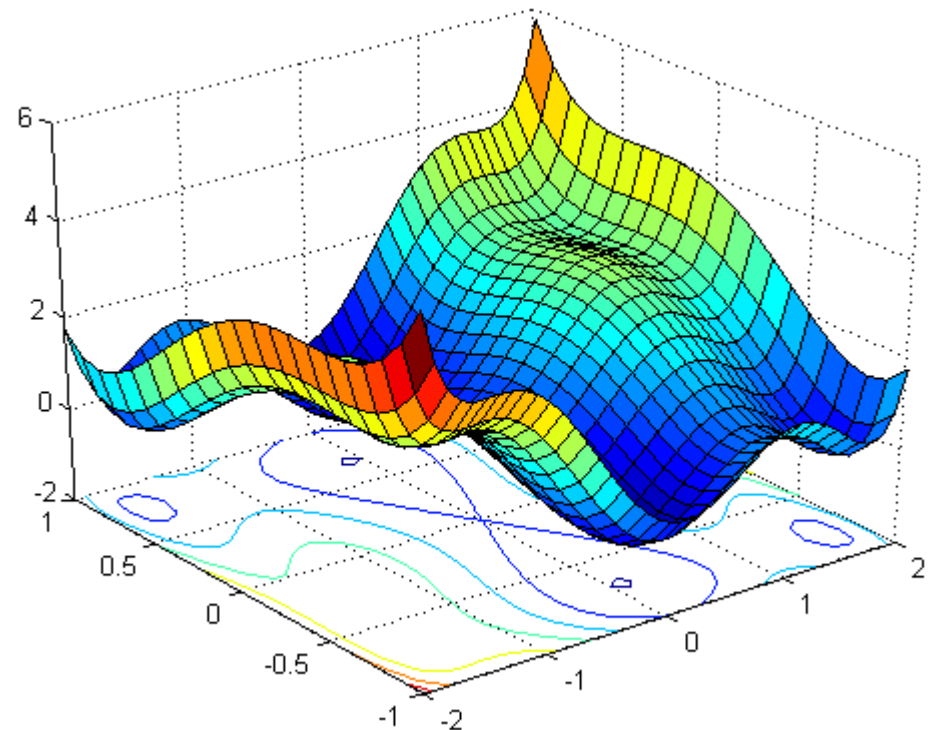
Input: 784 pixel values

Output: 10 numbers

Parameters: $784 \cdot 10 + 10 \cdot 10 = 7940$ weights plus 20 biases

Learning as Minimizing a Loss Function

This week we use the gradient descent algorithm to adapt the weights such that the loss function is minimized. For this we use again the direction of steepest descent (negative gradient) of a function.



Cost function:

Input: 7960 parameters (weights + biases)

Output: scalar (loss)

Training Neural Networks

Each **training example** has the form $\langle x_n, y_n \rangle$, where x_n is the vector of inputs, and y_n is the desired corresponding output vector.*

A **loss function** or **cost function** is a function that maps values of one or more variables onto a real number intuitively representing some "cost" associated with the event.

The **risk function R** is defined as the expectation of the loss functions over a dataset.

Backpropagation is a method used in artificial neural networks to calculate **gradients wrt. the risk function** that is then needed to calculate the weights and biases of the network.

In **(batched) gradient descent** the model is updated based on the **average of the gradients of the loss functions** calculated over all the training examples.

- Taking the average loss allows for the gradients to be calculated based on the overall performance of the model on the entire dataset, rather than being influenced by the specific characteristics of any one instance.
- Mini-batch gradient descent or SGD are used to speed up the process.

* We use n for an instance rather than the number of instances from now on to free some other indices.

Reminder: Chain Rule of (Univariate) Calculus

If g is differentiable at x and f is differentiable at $g(x)$, then the composite function $F(x) = f(g(x))$ is differentiable at x and F' is given by the product

$$F'(x) = f'(g(x)) * g'(x)$$

In Leibniz notation, if $y = f(u)$ and $u = g(x)$ are both differentiable functions, then

$$\frac{dy}{dx} = \frac{dy}{du} \frac{du}{dx}$$

Example:

$$\frac{d}{dx} \left((3x^2 - x + 1)^{\frac{1}{3}} \right) = \frac{1}{3} (3x^2 - x + 1)^{-\frac{2}{3}} (6x - 1)$$

Neural Networks and Gradient Descent

Minimizing the **empirical risk function** $R(\theta)$, which is modeling **expected loss** (as we don't know the true distribution of data). This means, the empirical risk $R(\theta)$ is the average loss over the training data.

$$R(\theta) = \mathbb{E}_{i \sim [1..n]} [L(y_n, g(\theta, x_n))] = \frac{1}{N} \sum_n L(y_n, g(\theta, x_n)) = \frac{1}{2N} \sum_n (y_n - g(\theta^T x_n))^2$$

derivative of $f(z)^2 \Rightarrow 2f(z)f'(z)$ (chain rule)

$$\nabla_{\theta} R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1)g'(\theta^T x_n)x_n = 0$$

$$g(z) = (1 + \exp(-z))^{-1}$$

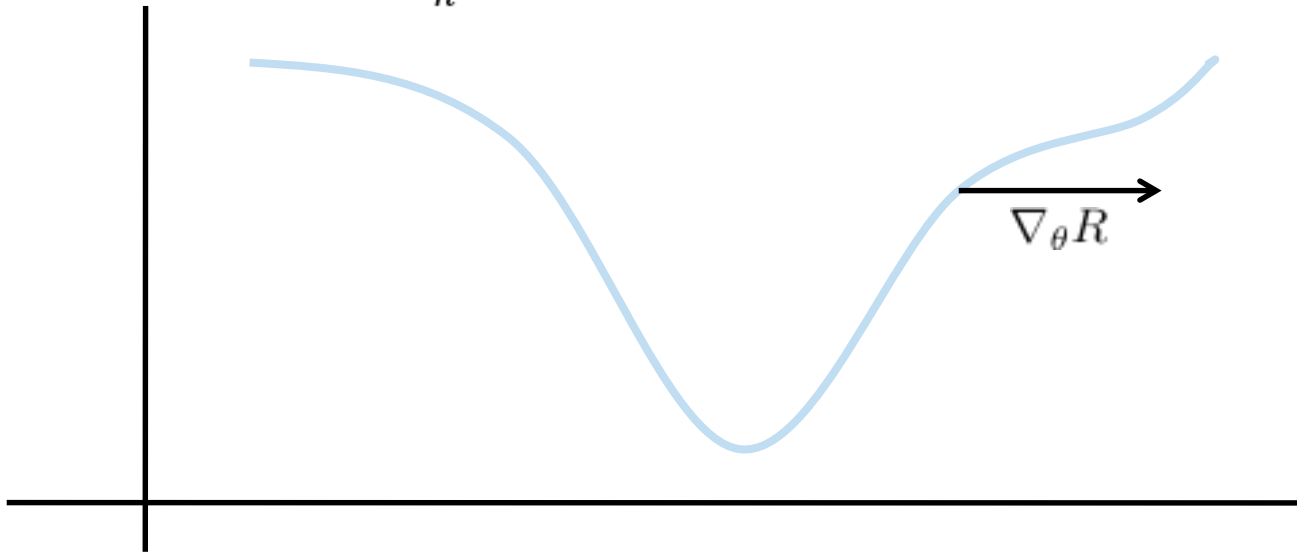
There is no “closed-form” solution. We use **gradient descent to minimize the empirical risk function** by iteratively updating the parameters of the model in the direction of the negative gradient.

Gradient Descent (1-Dimensional)

Gradient points in the direction of **steepest increase**

To **minimize** R , move in the **opposite direction**.

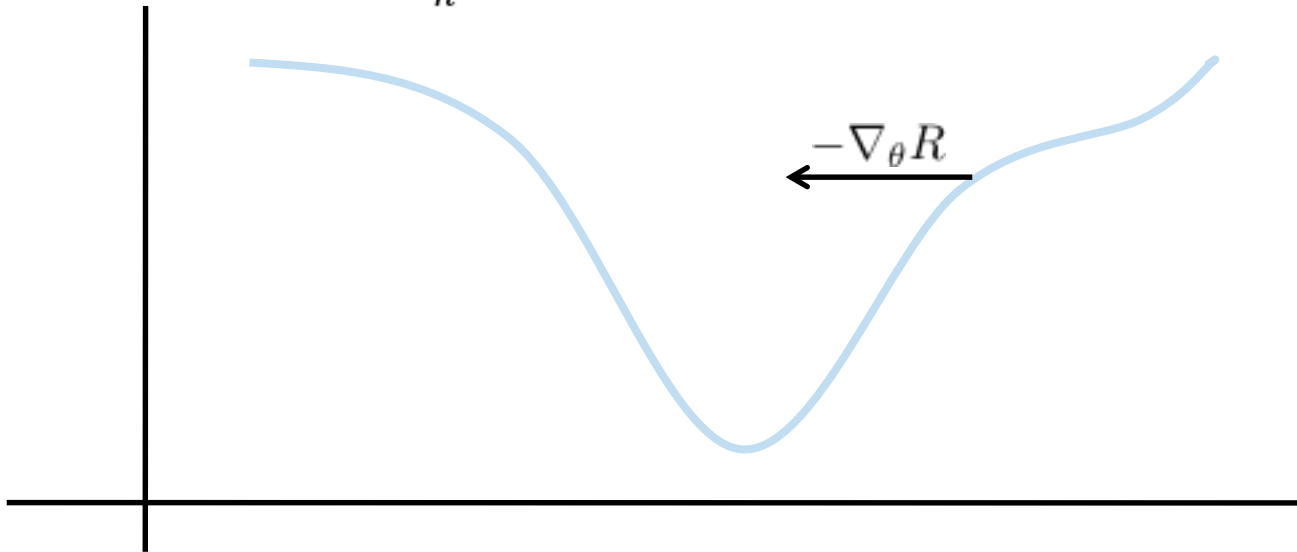
$$\nabla_{\theta} R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1)g'(\theta^T x_n)x_n = 0$$



Gradient Descent

Gradient points in the direction of steepest increase
To minimize R , move in the opposite direction.

$$\nabla_{\theta} R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1)g'(\theta^T x_n)x_n = 0$$



Gradient Descent

Initialize randomly

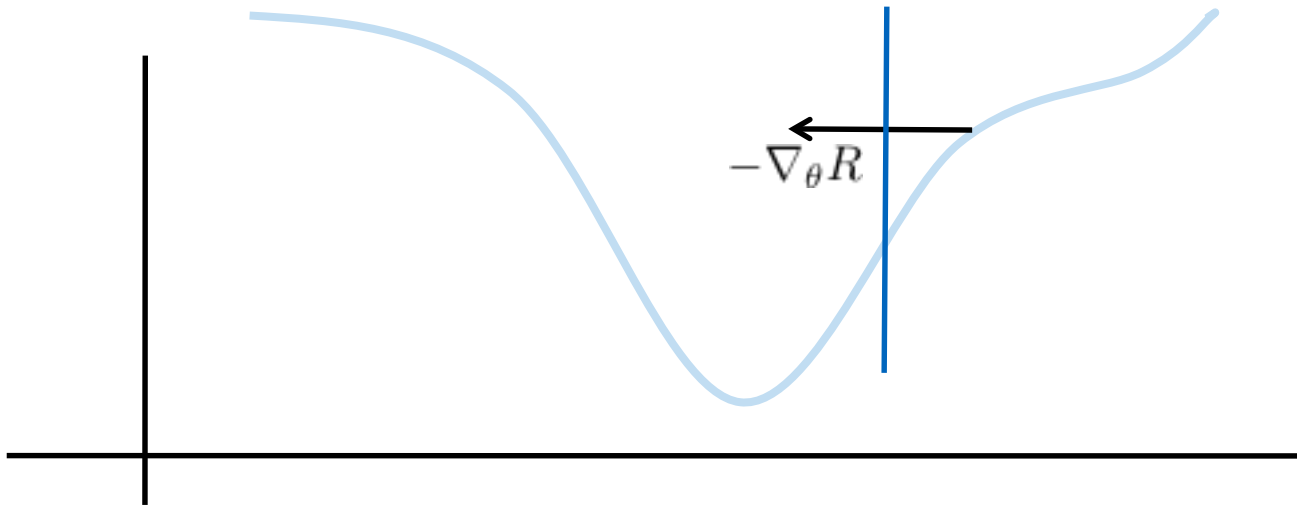
Update with small steps

(nearly) guaranteed to converge to the minimum

$$\theta_0 = \text{random}$$

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta} R \Big|_{\theta_t}$$

$$\nabla_{\theta} R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1)g'(\theta^T x_n)x_n = 0$$



Gradient Descent

Initialize randomly

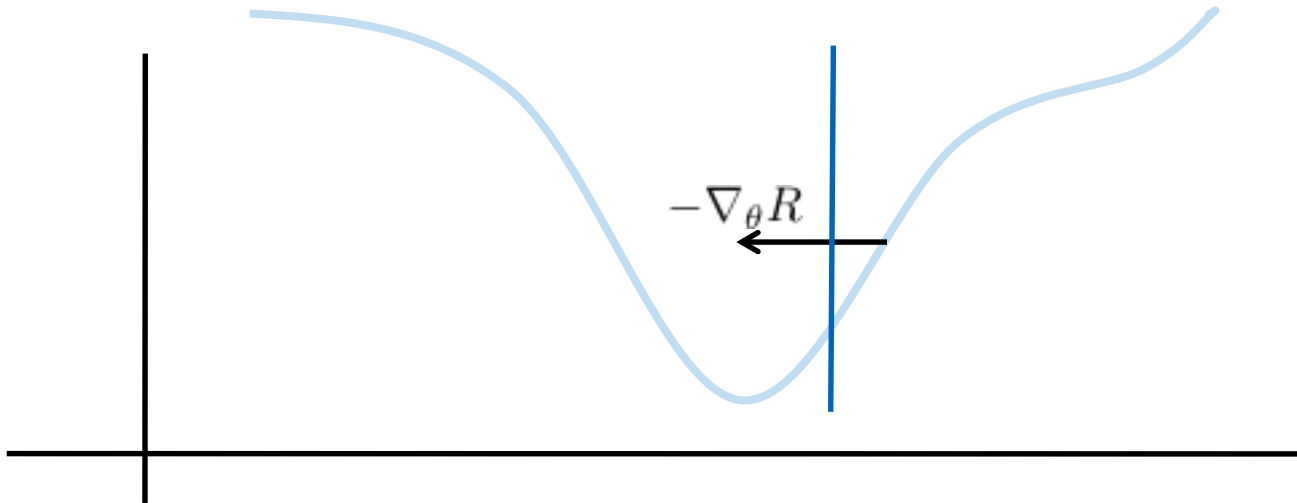
Update with small steps

(nearly) guaranteed to converge to the minimum

$$\theta_0 = \text{random}$$

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta} R \Big|_{\theta_t}$$

$$\nabla_{\theta} R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1)g'(\theta^T x_n)x_n = 0$$



Gradient Descent

Initialize randomly

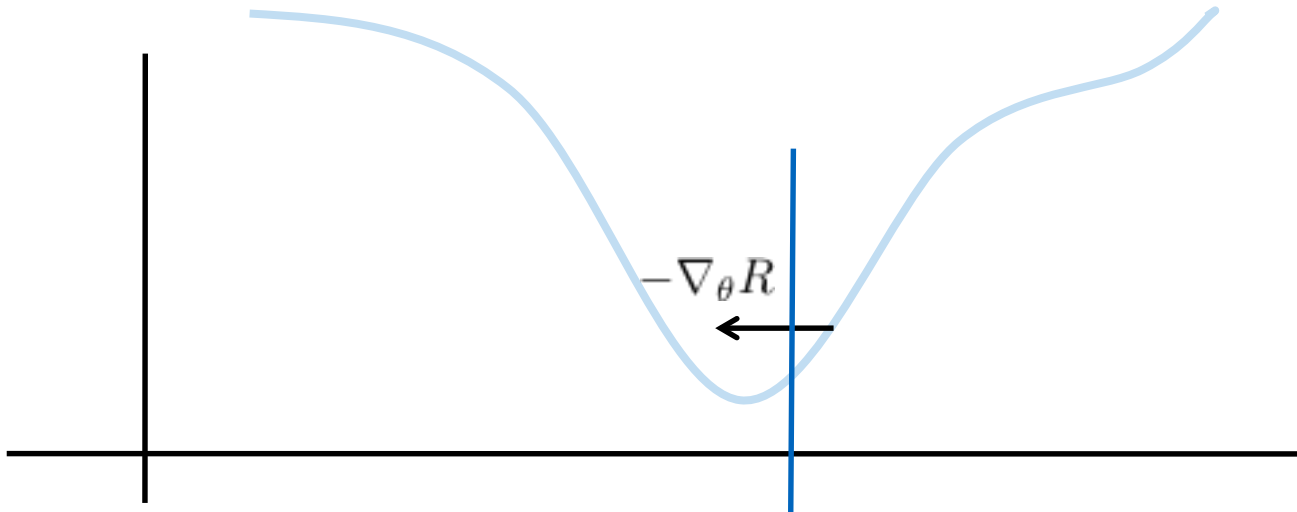
Update with small steps

(nearly) guaranteed to converge to the minimum

$$\theta_0 = \text{random}$$

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta} R \Big|_{\theta_t}$$

$$\nabla_{\theta} R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1)g'(\theta^T x_n)x_n = 0$$



Gradient Descent

Initialize randomly

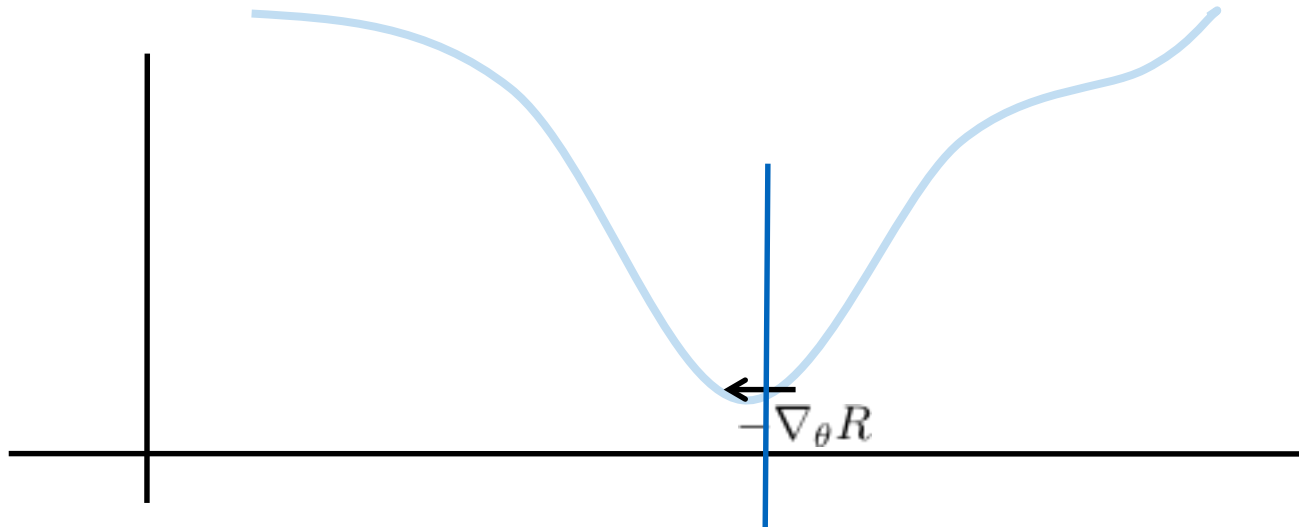
Update with small steps

(nearly) guaranteed to converge to the minimum

$$\theta_0 = \text{random}$$

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta} R \Big|_{\theta_t}$$

$$\nabla_{\theta} R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1)g'(\theta^T x_n)x_n = 0$$



Gradient Descent

Initialize randomly

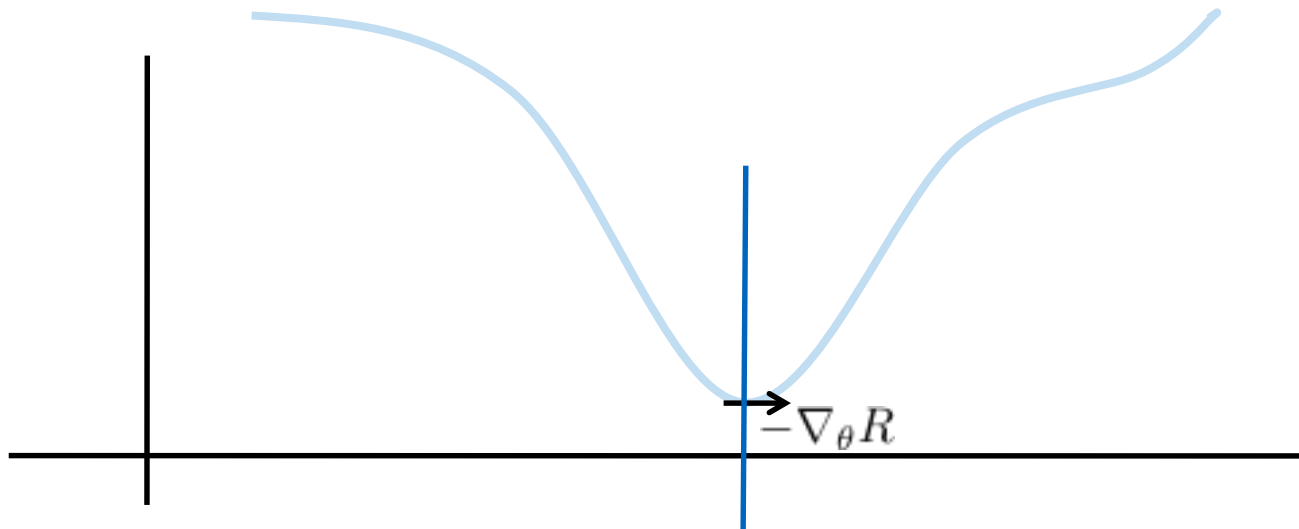
Update with small steps

(nearly) guaranteed to converge to the minimum

$$\theta_0 = \text{random}$$

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta} R \Big|_{\theta_t}$$

$$\nabla_{\theta} R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1)g'(\theta^T x_n)x_n = 0$$



Gradient Descent

Initialize randomly

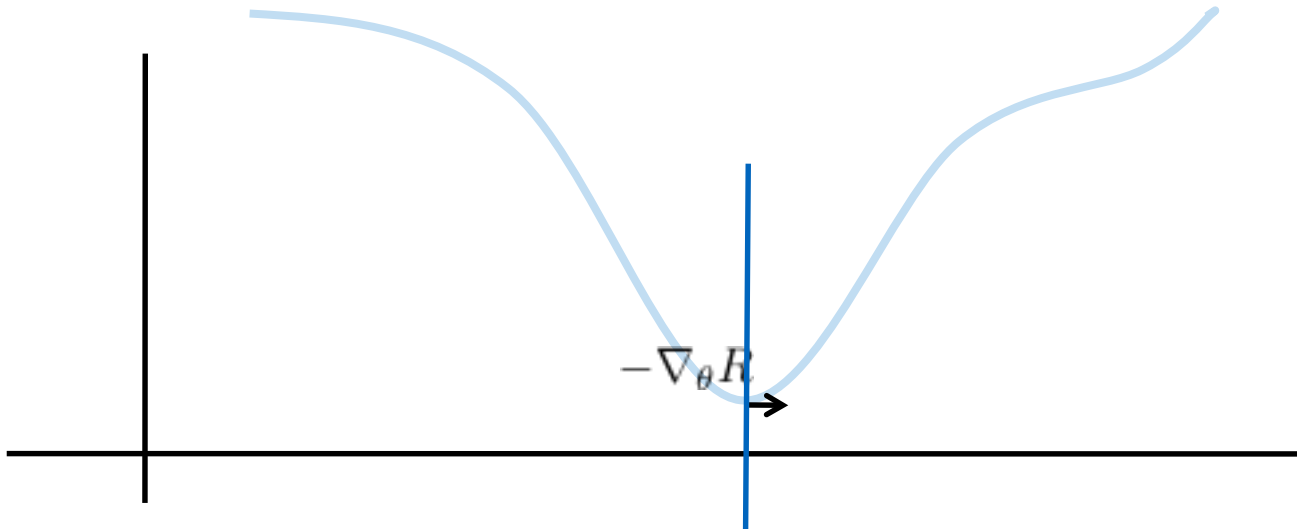
Update with small steps

(nearly) guaranteed to converge to the minimum

$$\theta_0 = \text{random}$$

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta} R \Big|_{\theta_t}$$

$$\nabla_{\theta} R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1)g'(\theta^T x_n)x_n = 0$$



Gradient Descent

Initialize randomly

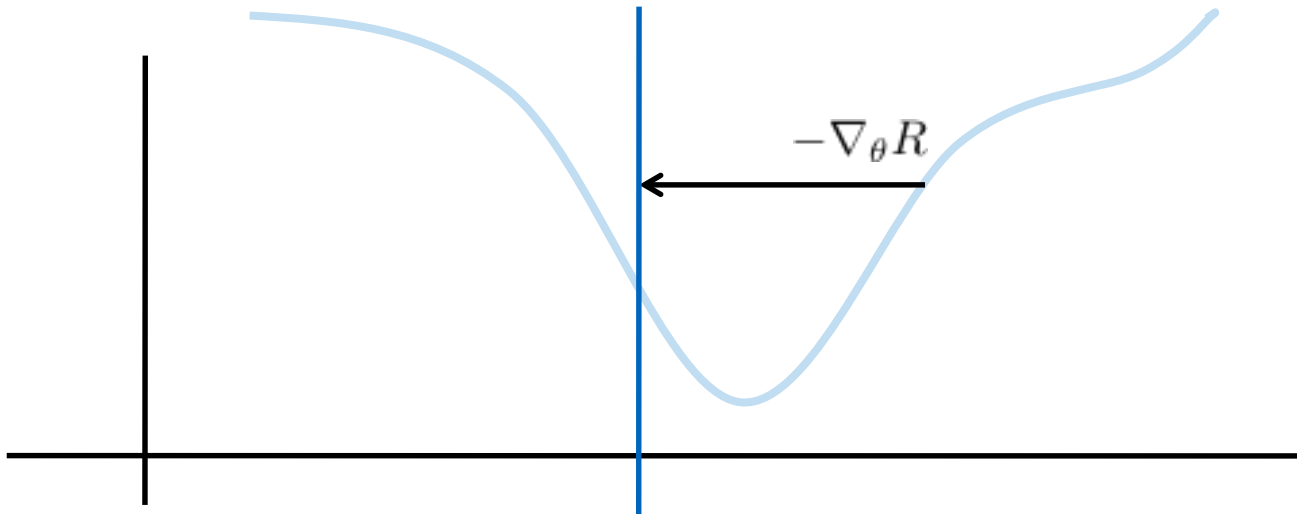
Update with small steps

Can oscillate if α is too large

$$\theta_0 = \text{random}$$

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta} R \Big|_{\theta_t}$$

$$\nabla_{\theta} R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1)g'(\theta^T x_n)x_n = 0$$



Gradient Descent

Initialize randomly

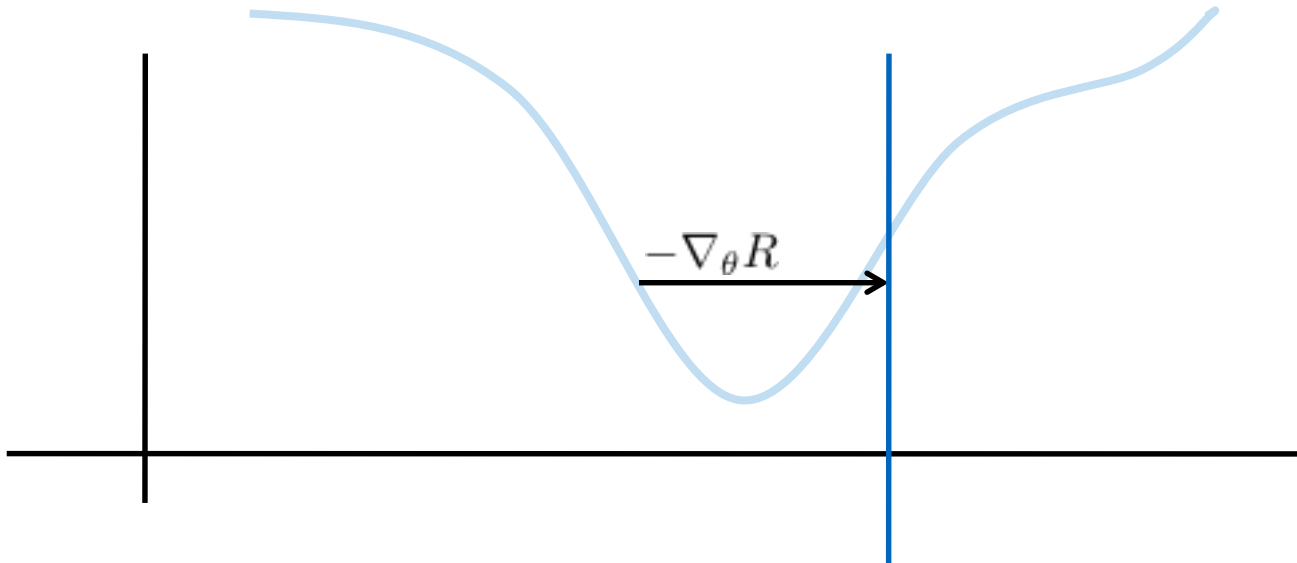
Update with small steps

Can oscillate if α is too large

$$\theta_0 = \text{random}$$

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta} R \Big|_{\theta_t}$$

$$\nabla_{\theta} R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1)g'(\theta^T x_n)x_n = 0$$



Gradient Descent

Initialize randomly

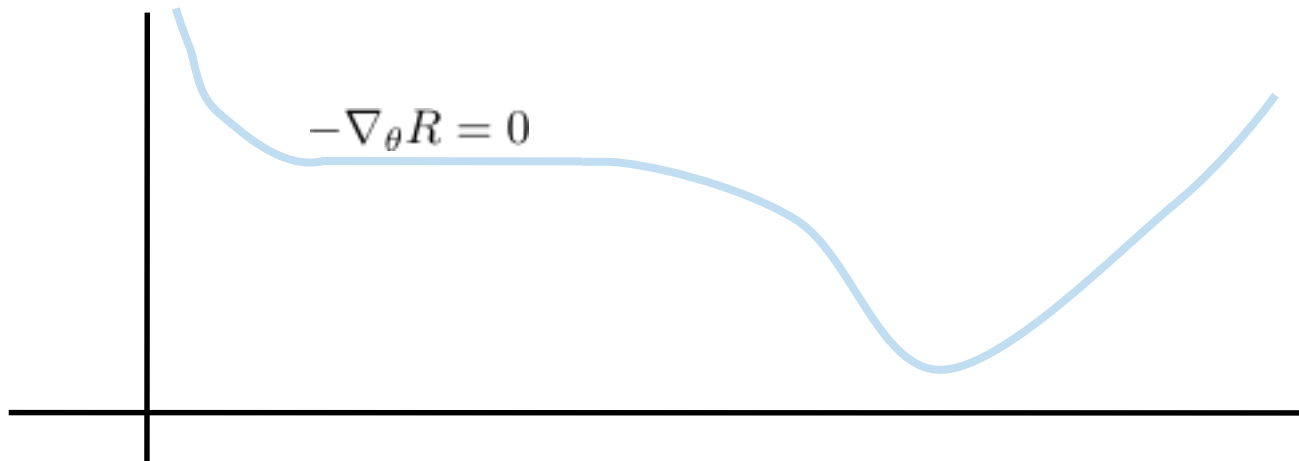
Update with small steps

Can stall if $-\nabla_{\theta} R$ is ever 0 not at the minimum

$$\theta_0 = \text{random}$$

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta} R \Big|_{\theta_t}$$

$$\nabla_{\theta} R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1)g'(\theta^T x_n)x_n = 0$$



What distinguishes a feed-forward NN from a logistic regression?



Local Minima of Risk Functions

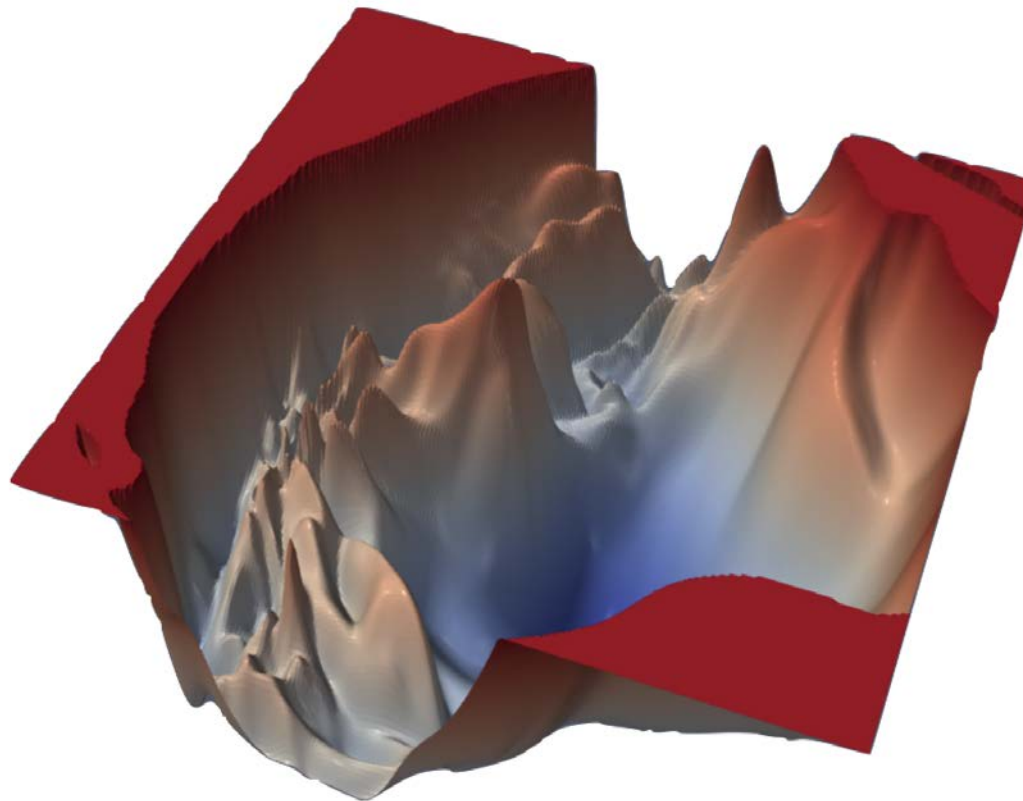
The **risk function** of a neural network is in **general neither convex nor concave**. This means that the Hessian is neither positive semidefinite, nor negative semidefinite.

The probability of finding a "bad" (high value) local minimum is non-zero for small-size neural networks and decreases quickly with network size.

For large networks, most local minima are equivalent and yield similar performance on a test set

- Choromanska, Anna, et al. "The loss surfaces of multilayer networks." *Artificial intelligence and statistics*. PMLR, 2015. <https://arxiv.org/pdf/1412.0233v3.pdf>.
- If there's no hidden layers the logistic neural network is convex, just like logistic regression.
- Neural networks with linear activation functions and square loss also yield a convex problem.

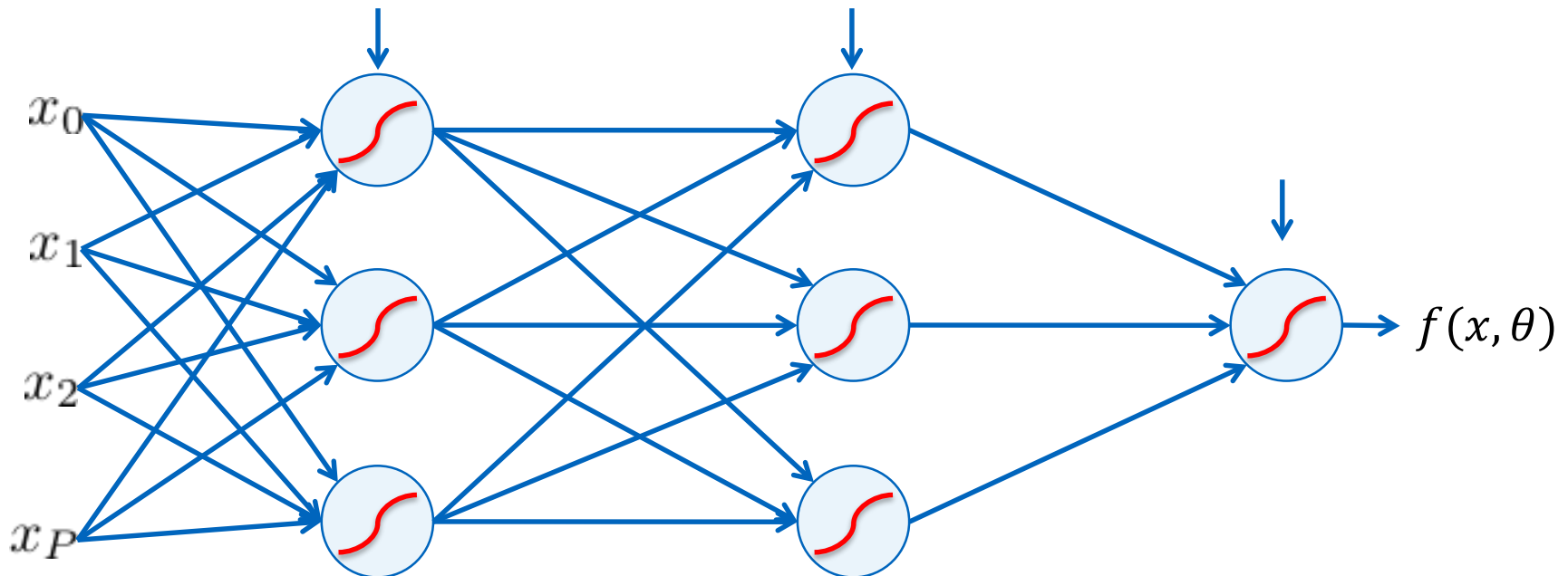
Loss Landscapes in Neural Networks



Li, Hao, et al. "Visualizing the loss landscape of neural nets." *Advances in neural information processing systems* 31 (2018).

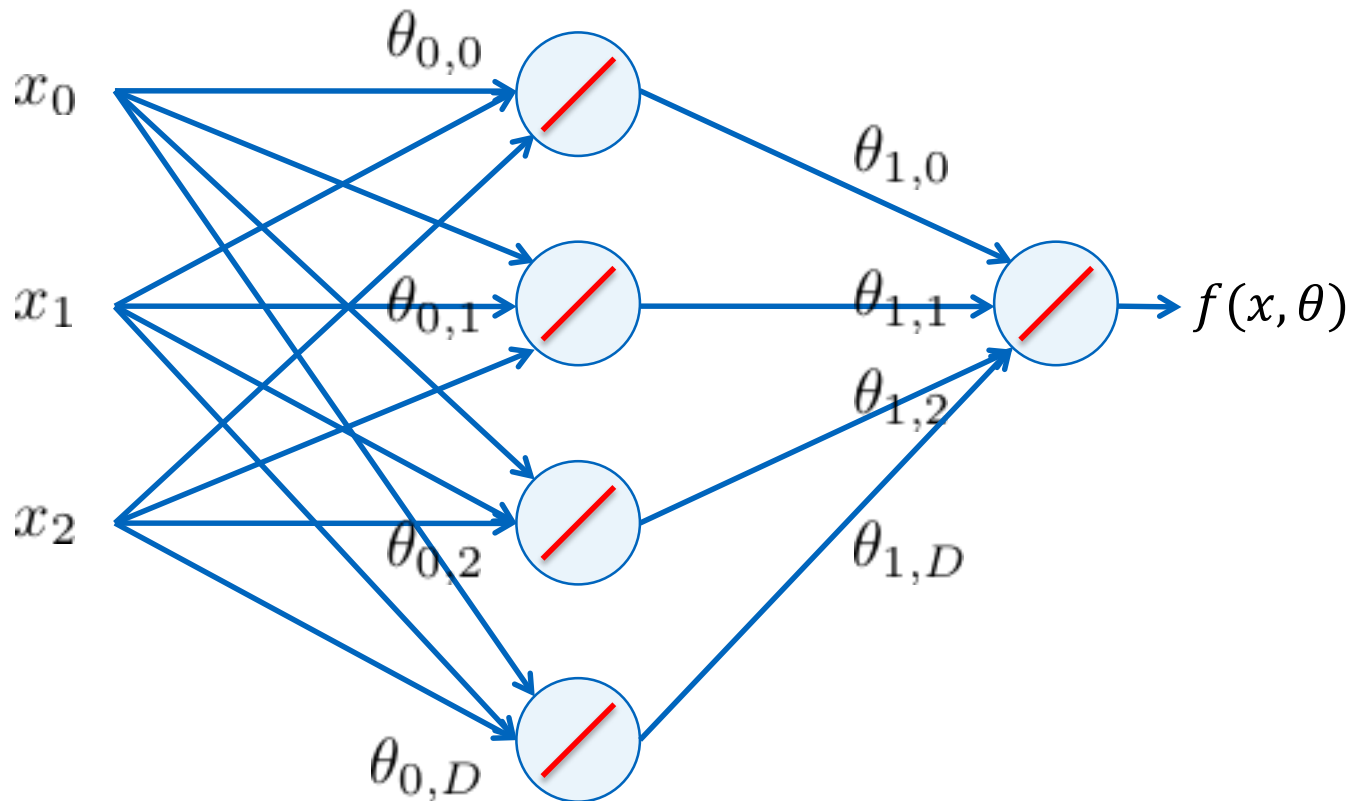
Multilayer Networks

- P input variables and N data instances. Cascade neurons together.
- The output from one layer is the input to the next.
- Each layer has its own sets of weights.



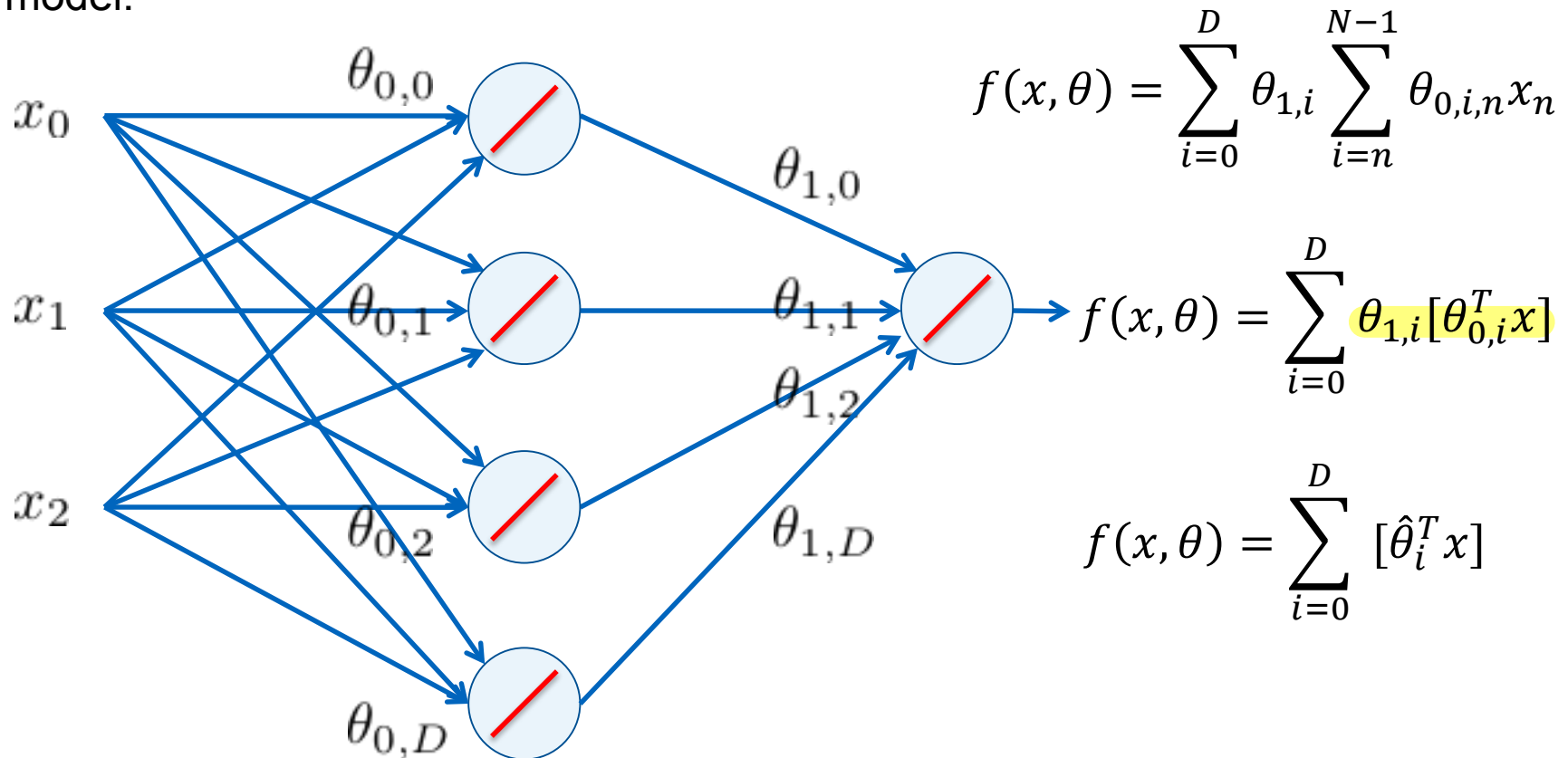
Linear Regression Neural Networks

What happens when we arrange **linear neurons** in a multilayer network?



Linear Regression Neural Networks

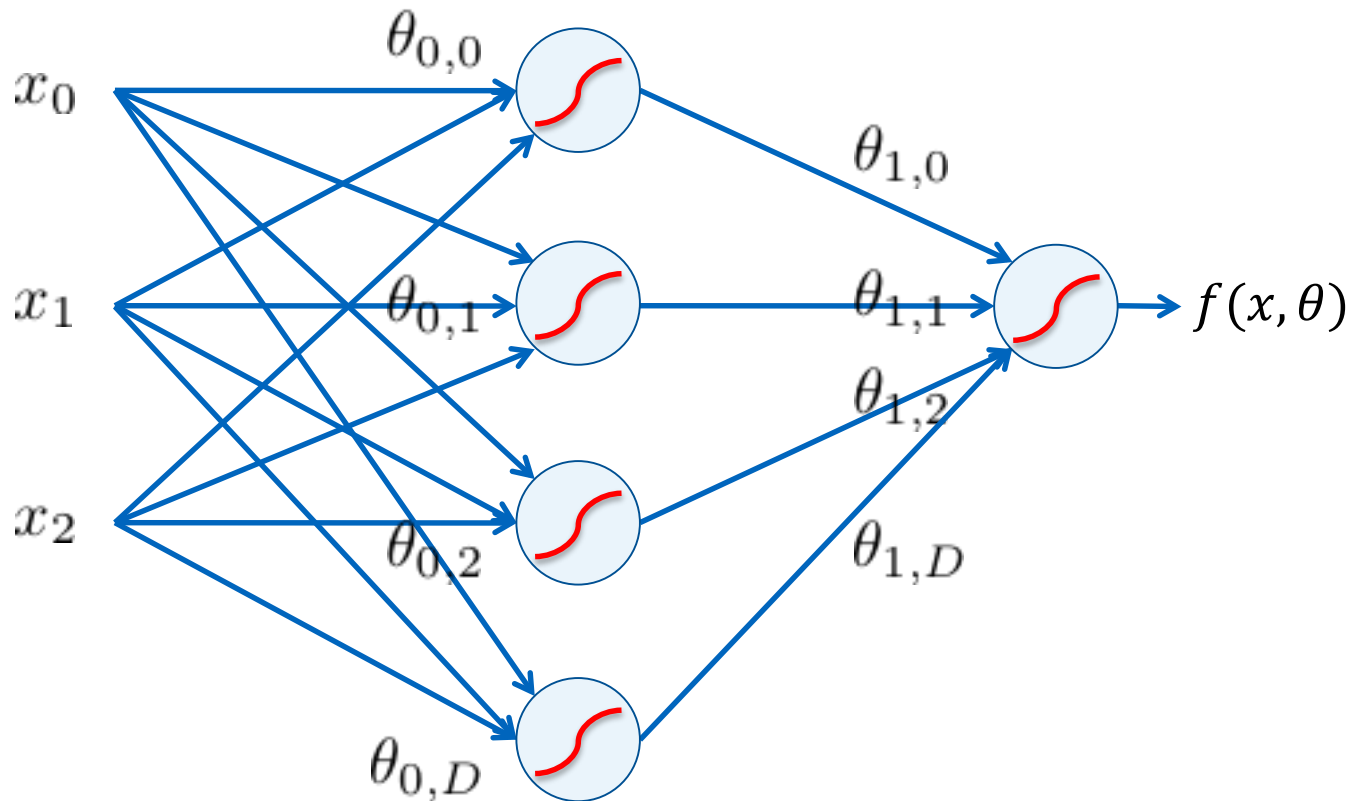
The composition of two linear transformations is itself a linear transformation:
A **neural network** with a **linear activation** function is simply a linear regression model.



Neural Networks

We want to introduce non-linearities to the network.

Non-linearities allow a network to identify complex regions in space.

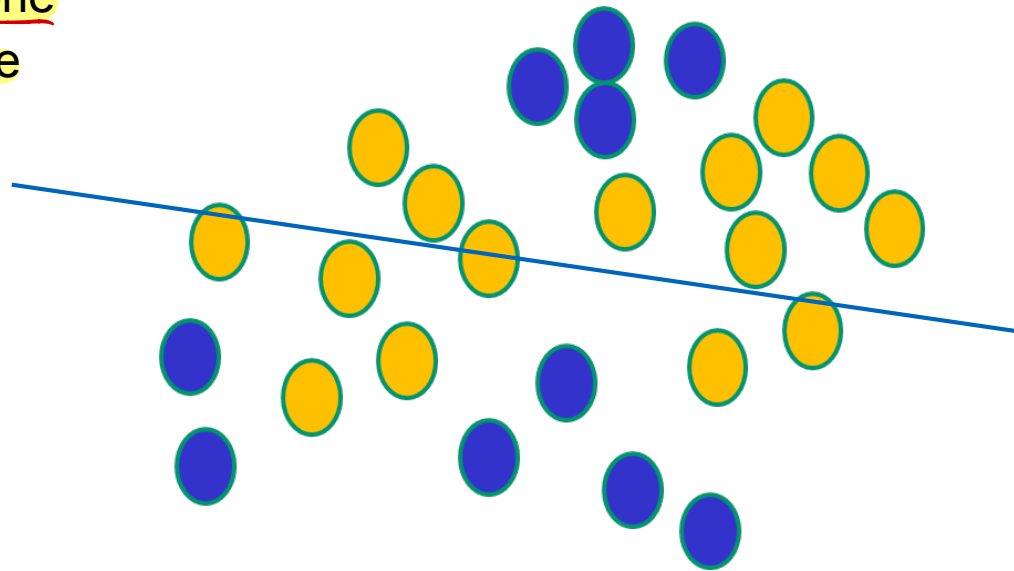


Linearly Separability

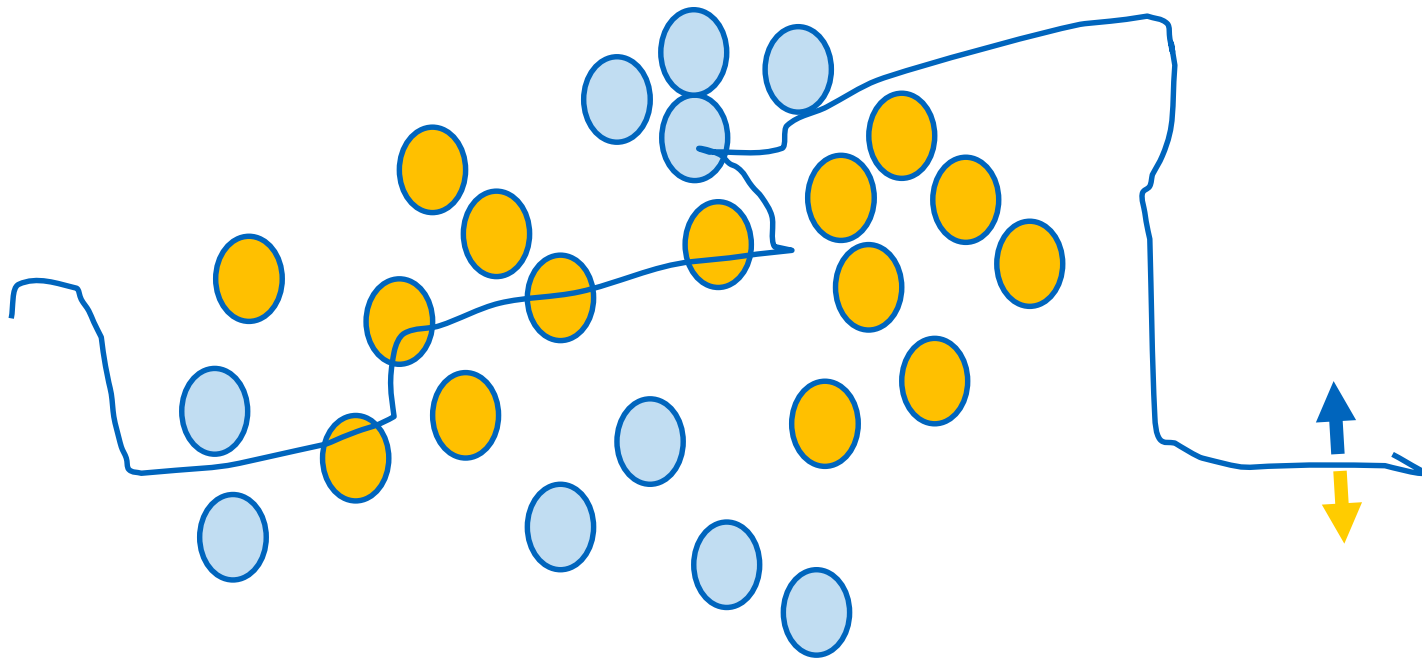
Two classes of points are **linearly separable**, iff there exists a line such that all the points of one class fall on one side of the line, and all the points of the other class fall on the other side of the line.

If $f(x)$ is linear, the NN can **only** draw straight decision boundaries (even if there are many layers of units).

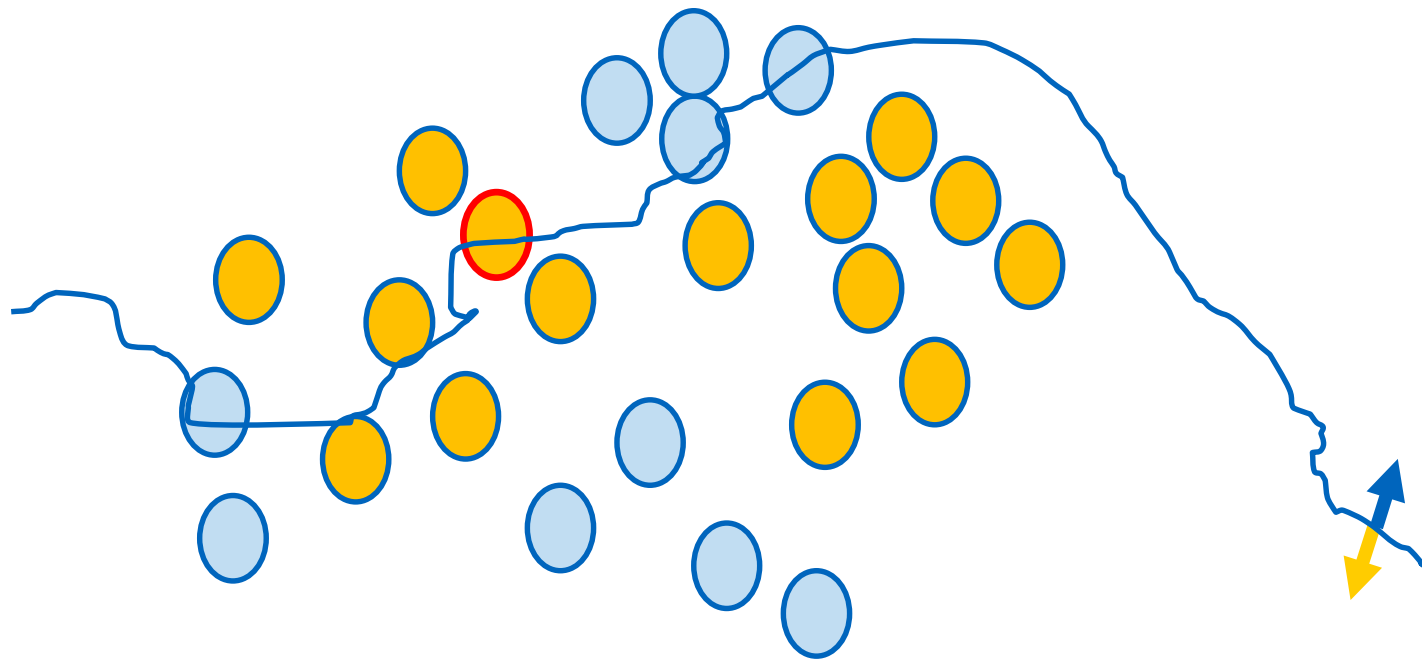
Established approach is to use the non-linear, differentiable sigmoidal “logistic” function $f(x)$, that we have seen earlier, which can draw complex boundaries.



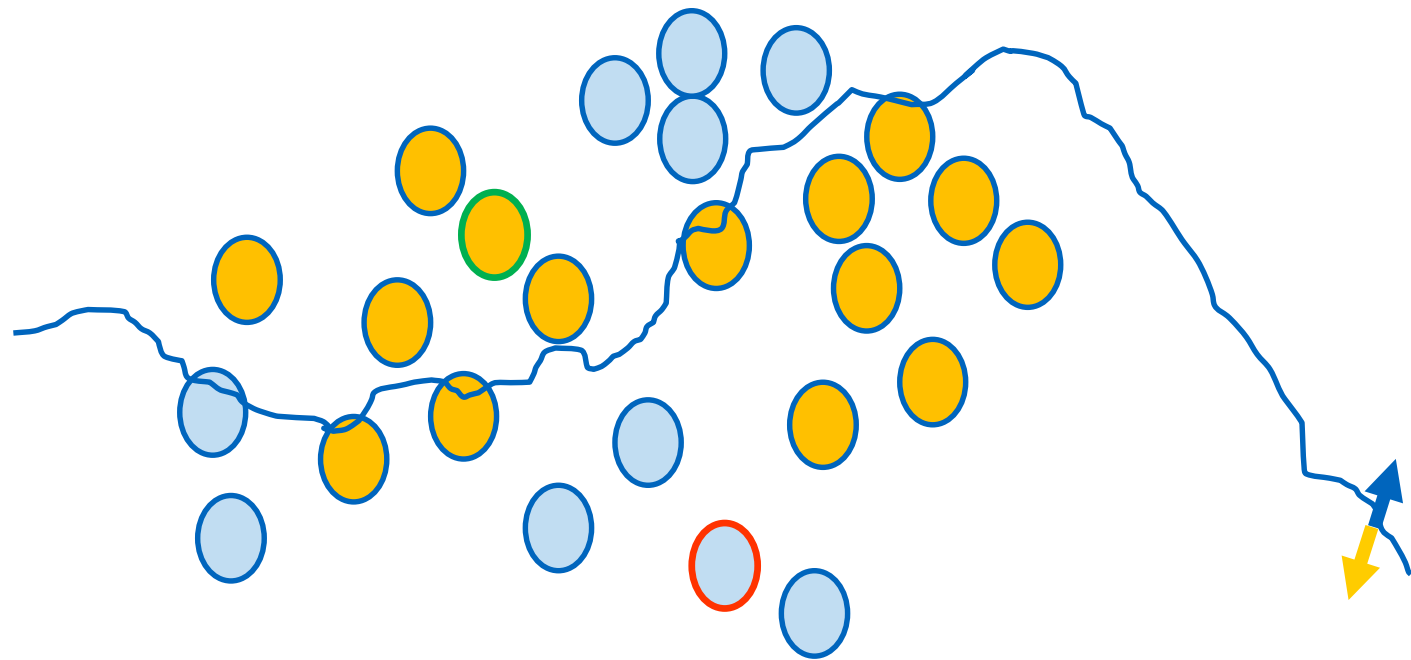
Initial Weights



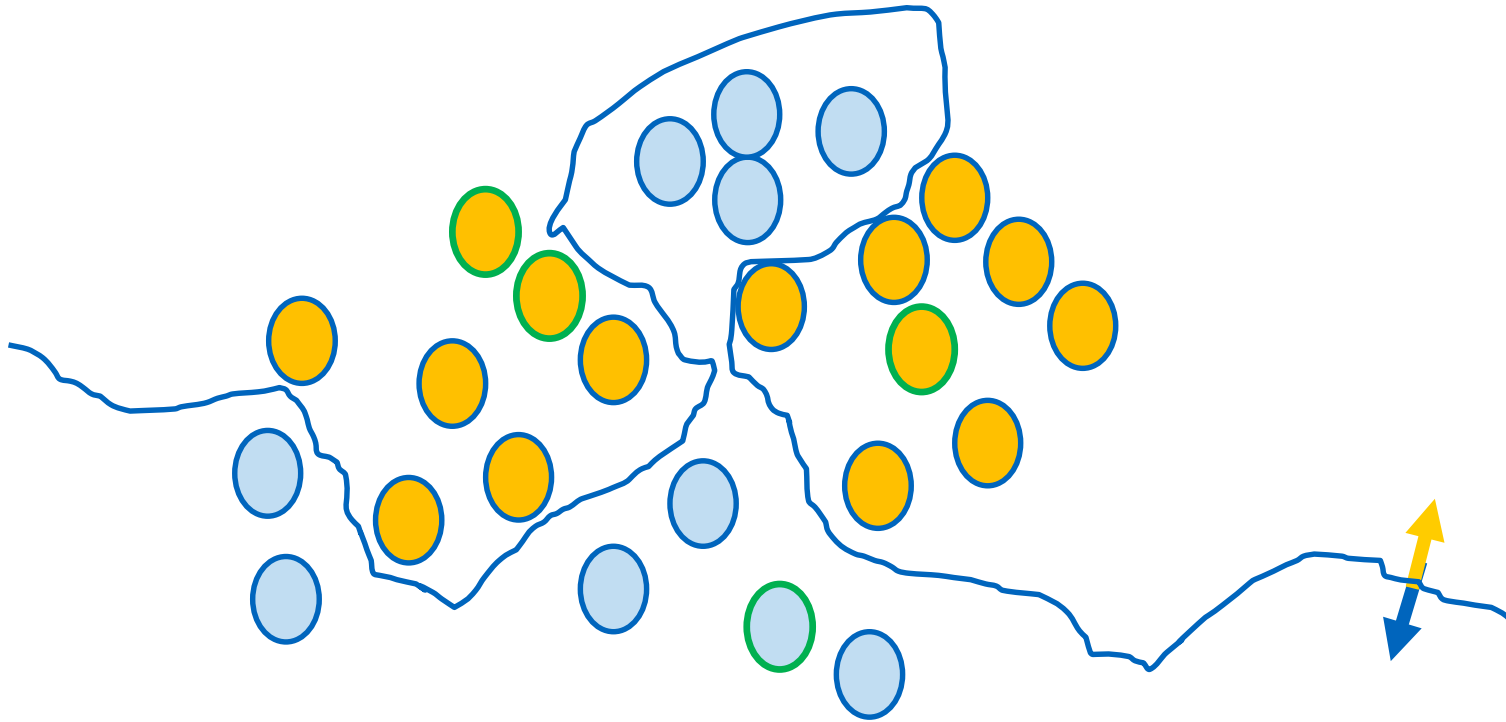
Present Instance and Adjust Weights



Present Instance and Adjust Weights



Eventual Final Classification



NNs are making thousands of tiny adjustments, each making the network do better at the most recent pattern, but perhaps a little worse on many others.

Neural Networks Recap

Number of Nodes per Layer

Input layer:

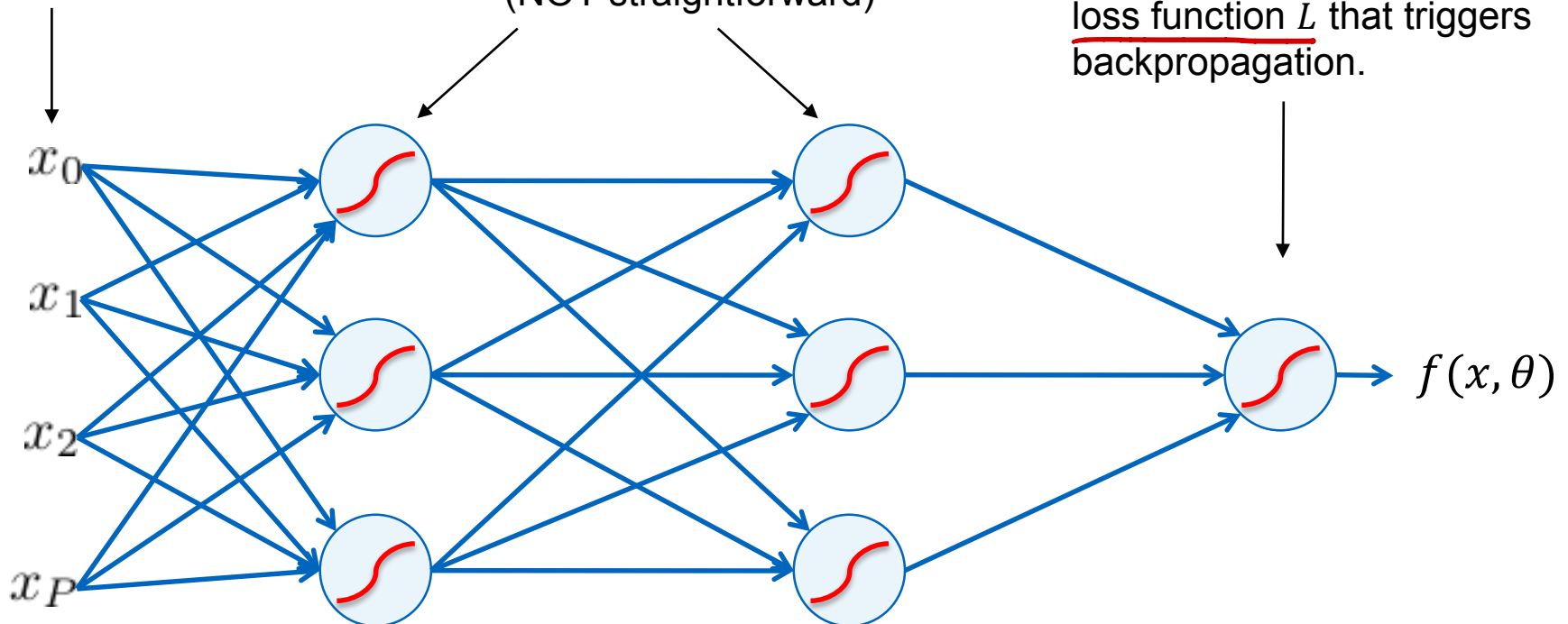
depending on
data dimension

Hidden Layers:

Should be chosen in a
way that improves
model performance.
(NOT straightforward)

Output layer:

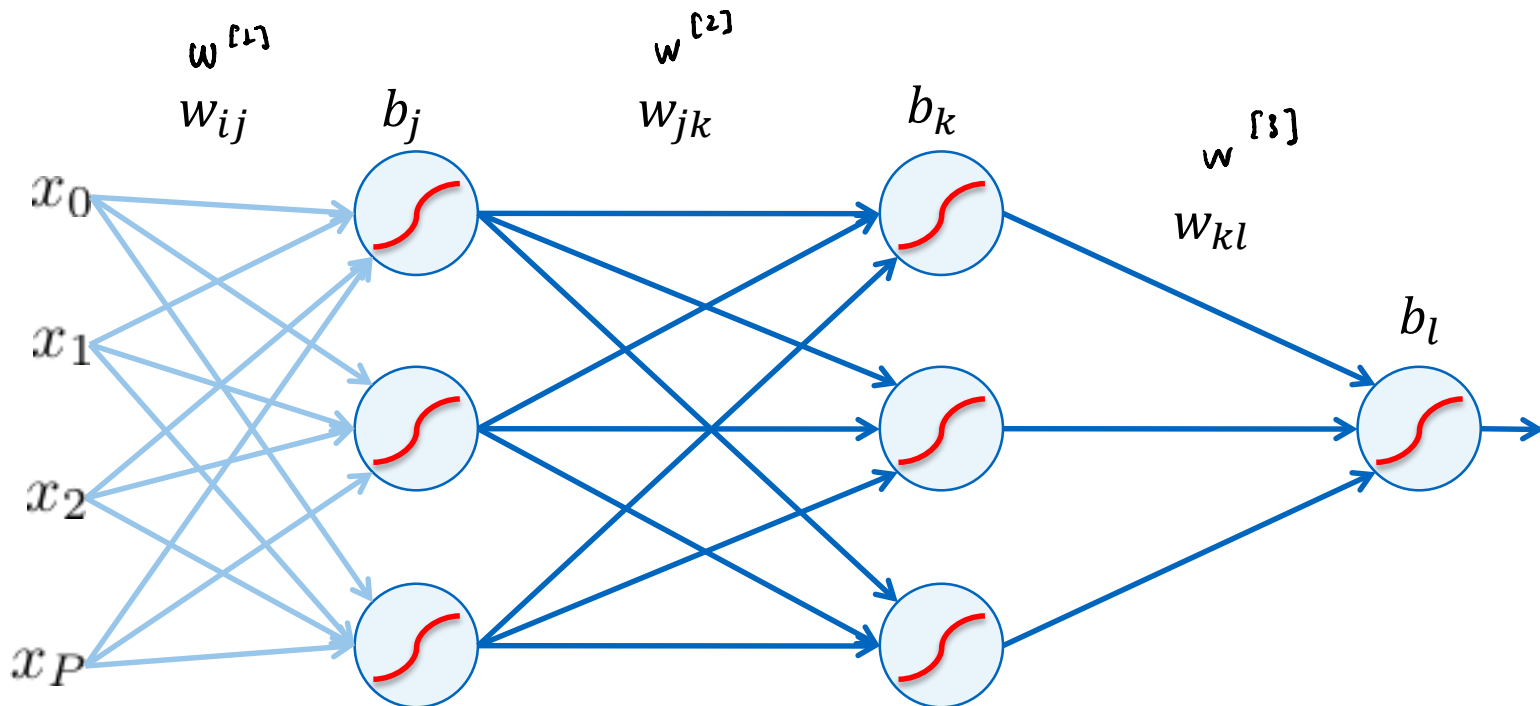
Corresponding to modeling
needs, i.e. 1 for binary
classification, $k - 1$ nodes
for multiclass classification,
etc. Activation of output layer
for each observation
becomes the input to the
loss function L that triggers
backpropagation.



Feed-Forward Networks

Instances are fed forward through the network.

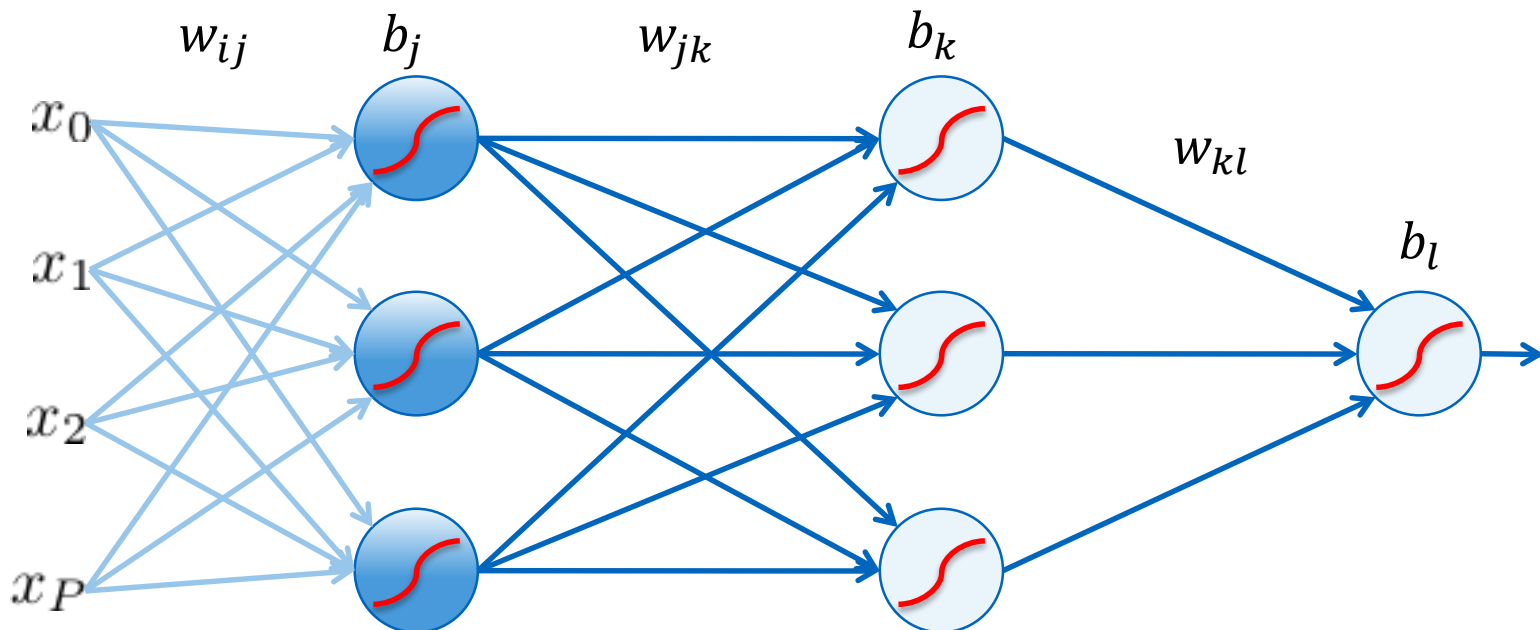
Rewrite θ as $\theta = \{w_{ij}, b_j, w_{jk}, b_k, w_{kl}, b_l\}$.
 $= \{w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, w^{[3]}, b^{[3]}\}$



Feed-Forward Networks

Instances are fed forward through the network.

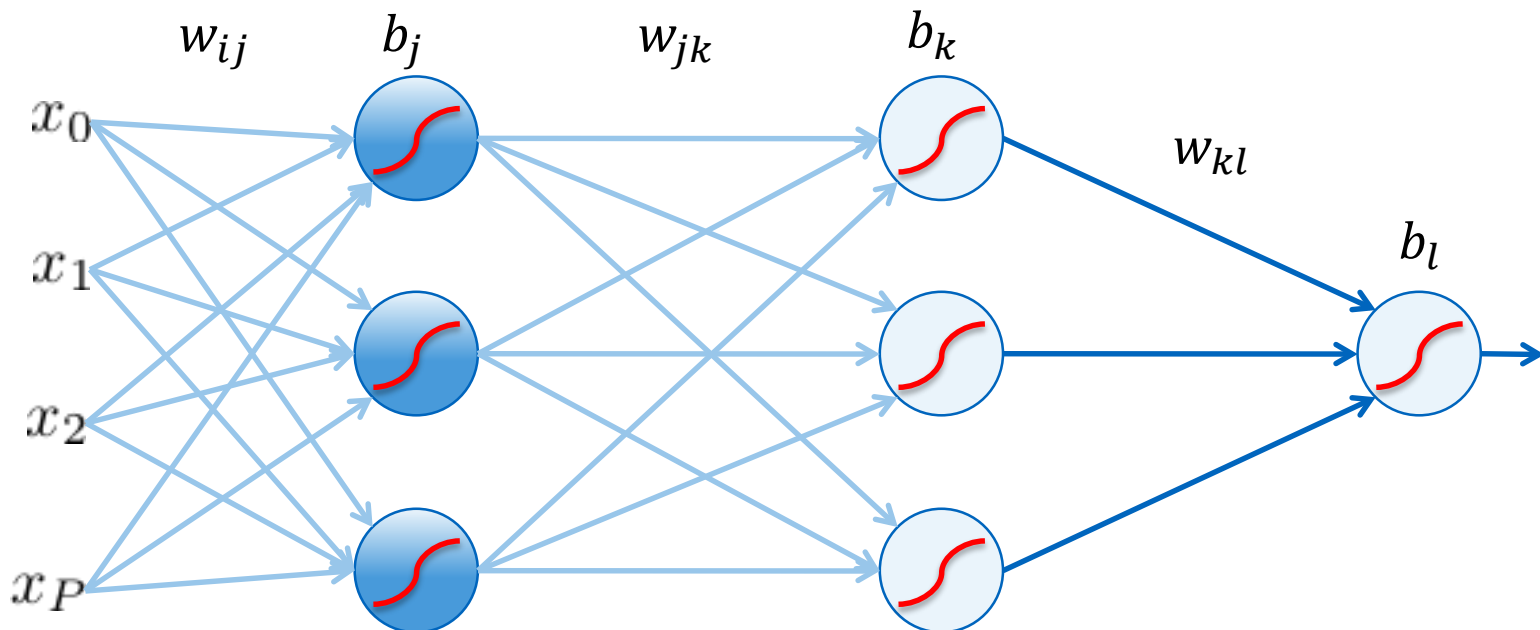
Rewrite θ as $\theta = \{w_{ij}, b_j, w_{jk}, b_k, w_{kl}, b_l\}$.



Feed-Forward Networks

Instances are fed forward through the network.

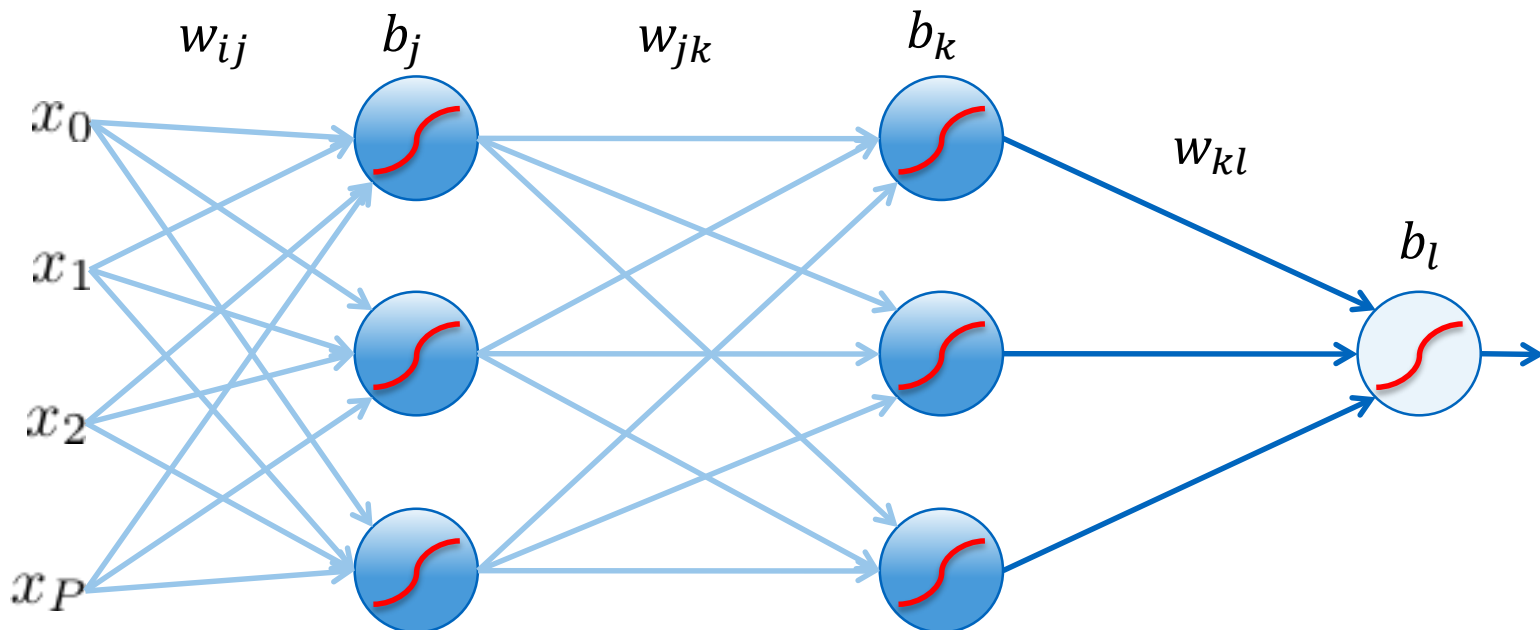
Rewrite θ as $\theta = \{w_{ij}, b_j, w_{jk}, b_k, w_{kl}, b_l\}$.



Feed-Forward Networks

Instances are fed forward through the network.

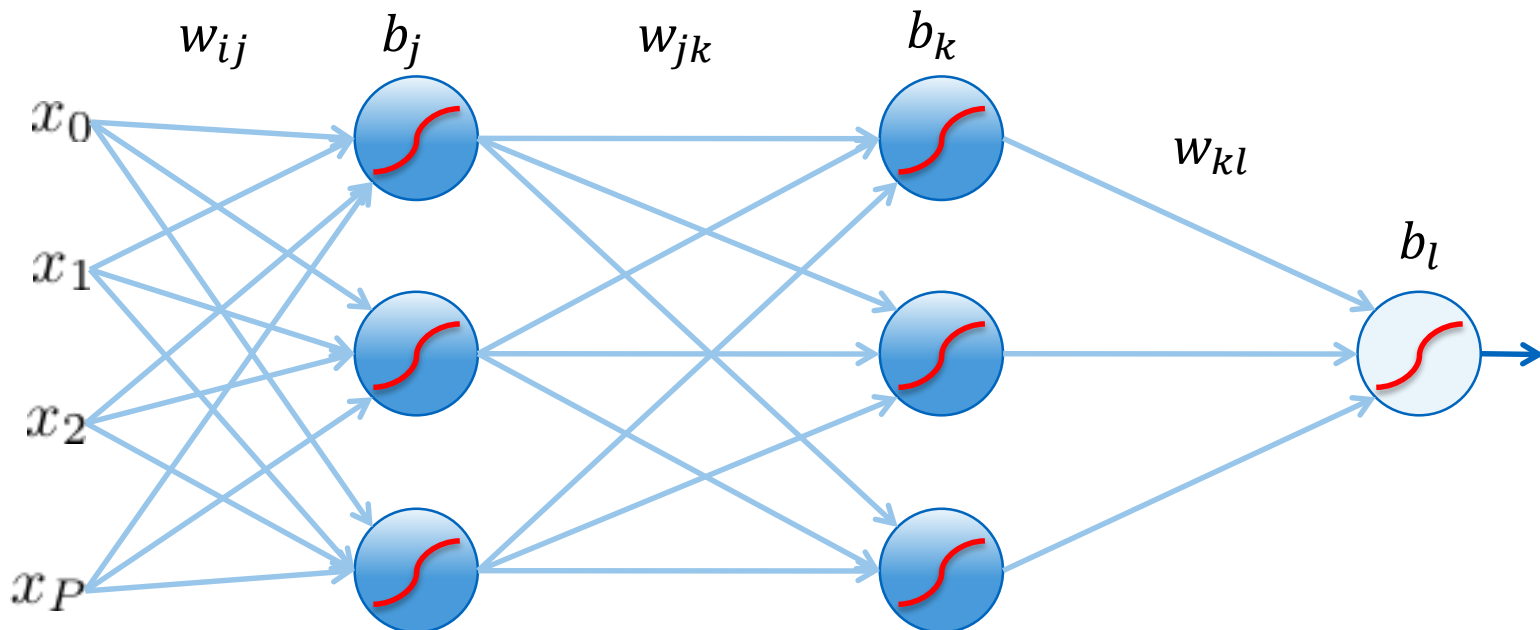
Rewrite θ as $\theta = \{w_{ij}, b_j, w_{jk}, b_k, w_{kl}, b_l\}$.



Feed-Forward Networks

Instances are fed forward through the network.

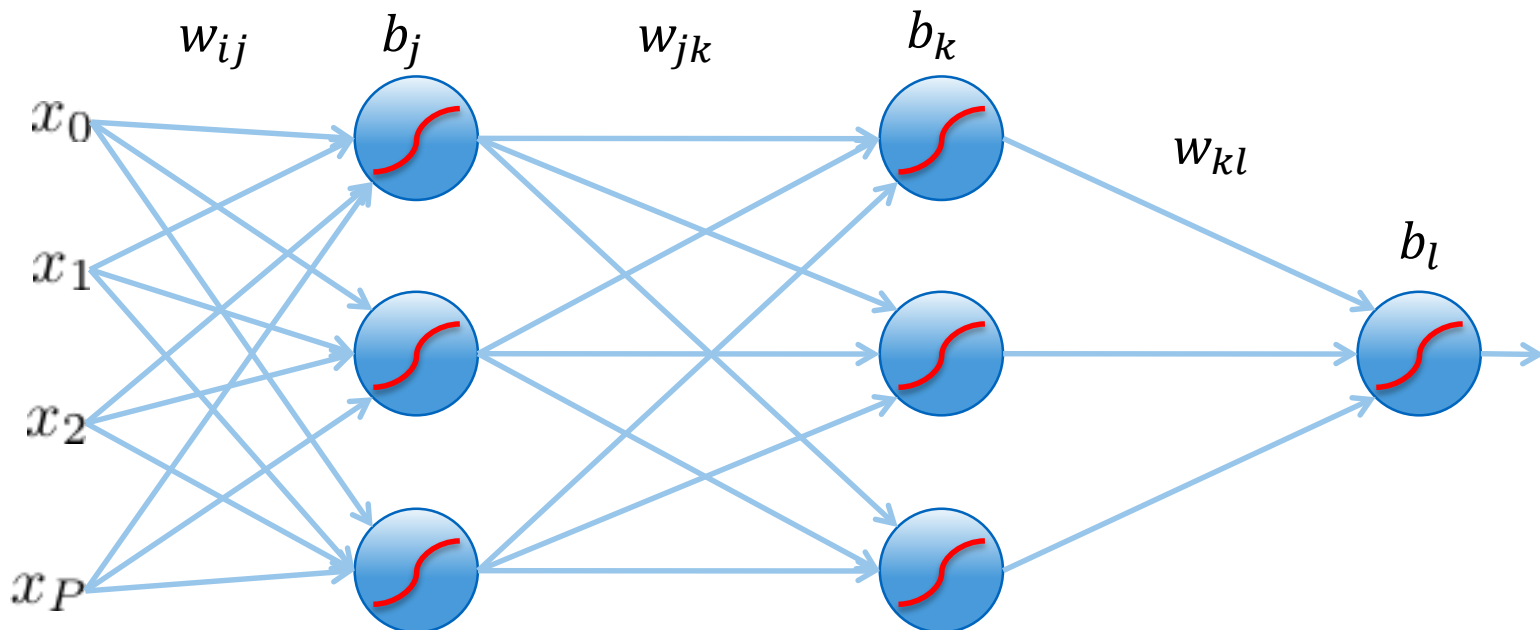
Rewrite θ as $\theta = \{w_{ij}, b_j, w_{jk}, b_k, w_{kl}, b_l\}$.



Feed-Forward Networks

Instances are fed forward through the network.

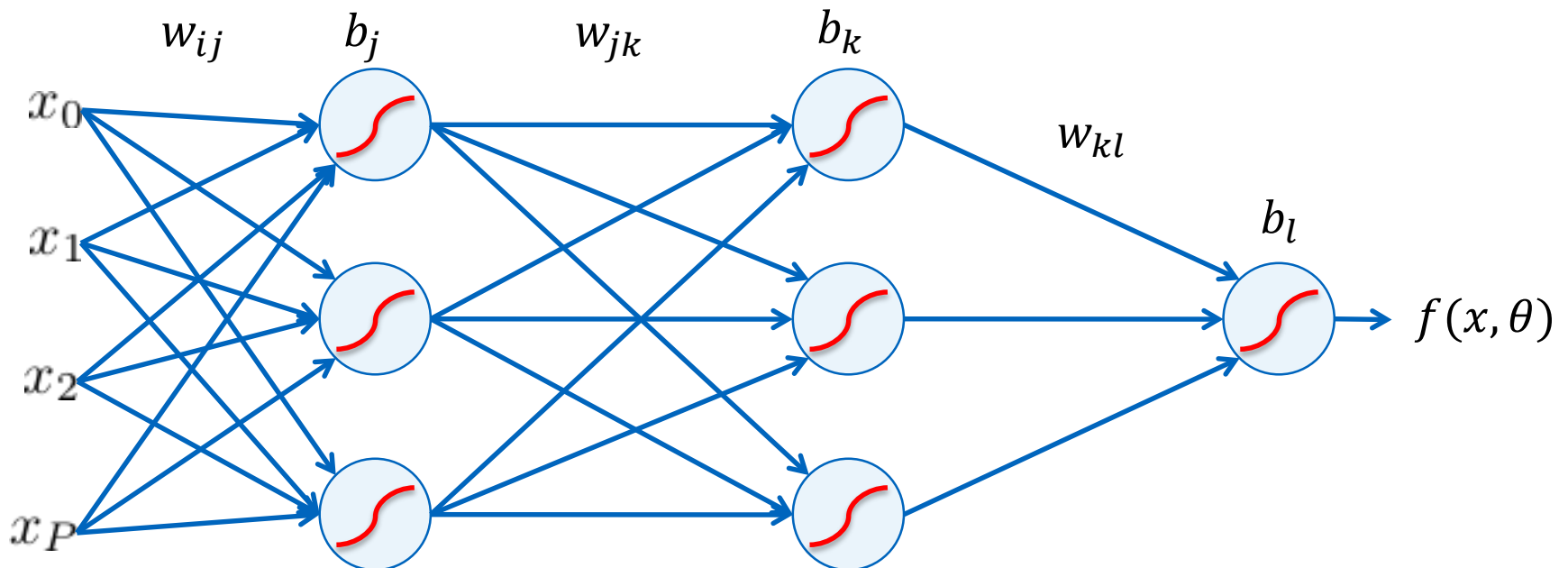
Rewrite θ as $\theta = \{w_{ij}, b_j, w_{jk}, b_k, w_{kl}, b_l\}$.



Error Backpropagation

We will do gradient descent on the whole network.

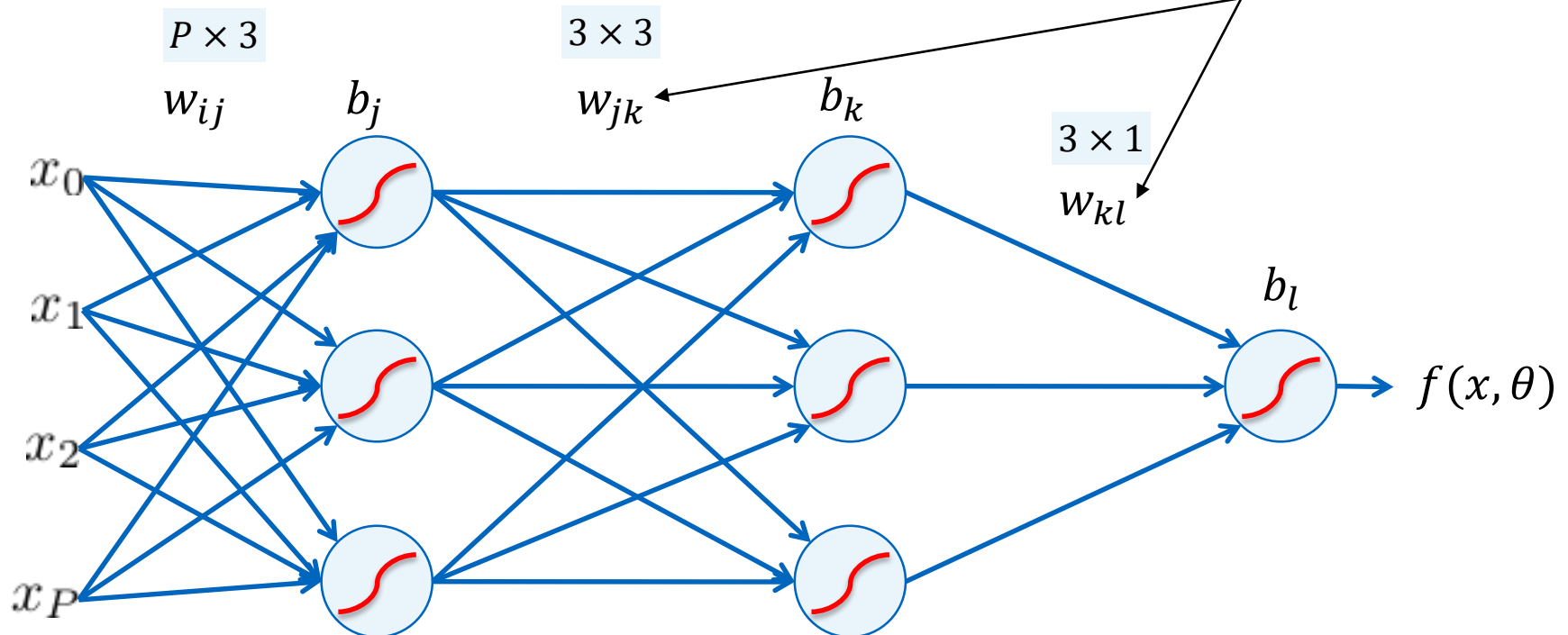
Training will proceed from the last layer to the first.



Error Backpropagation

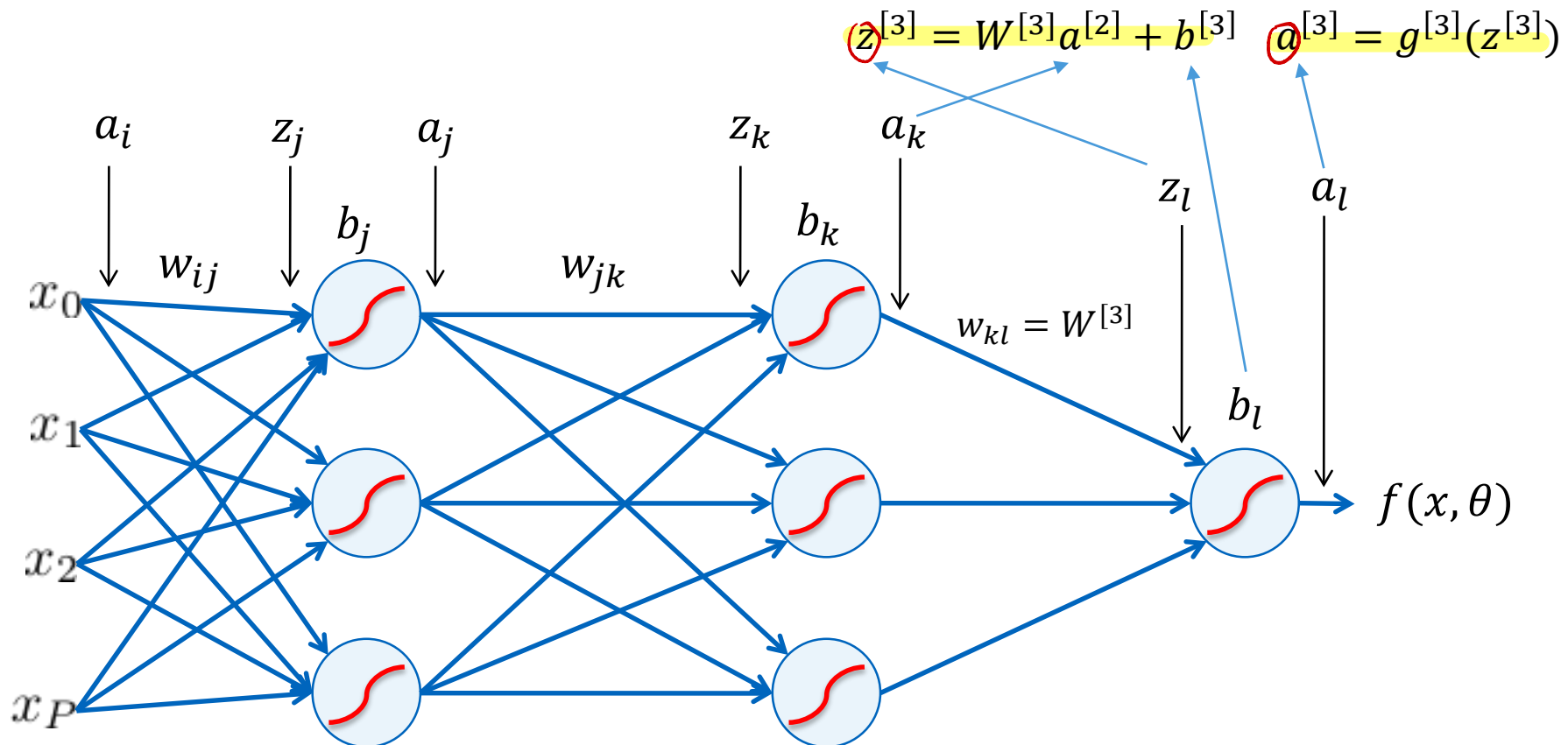
Backpropagation is used to calculate the gradients of the parameters of a neural network with respect to the loss function. Let's introduce some variables:

Note that w_{kl} refers to a 3×1 matrix. It is sometimes more convenient to write this as the weights of the last layer: $W^{[3]}$ or $W^{[2]}$.



Error Backpropagation

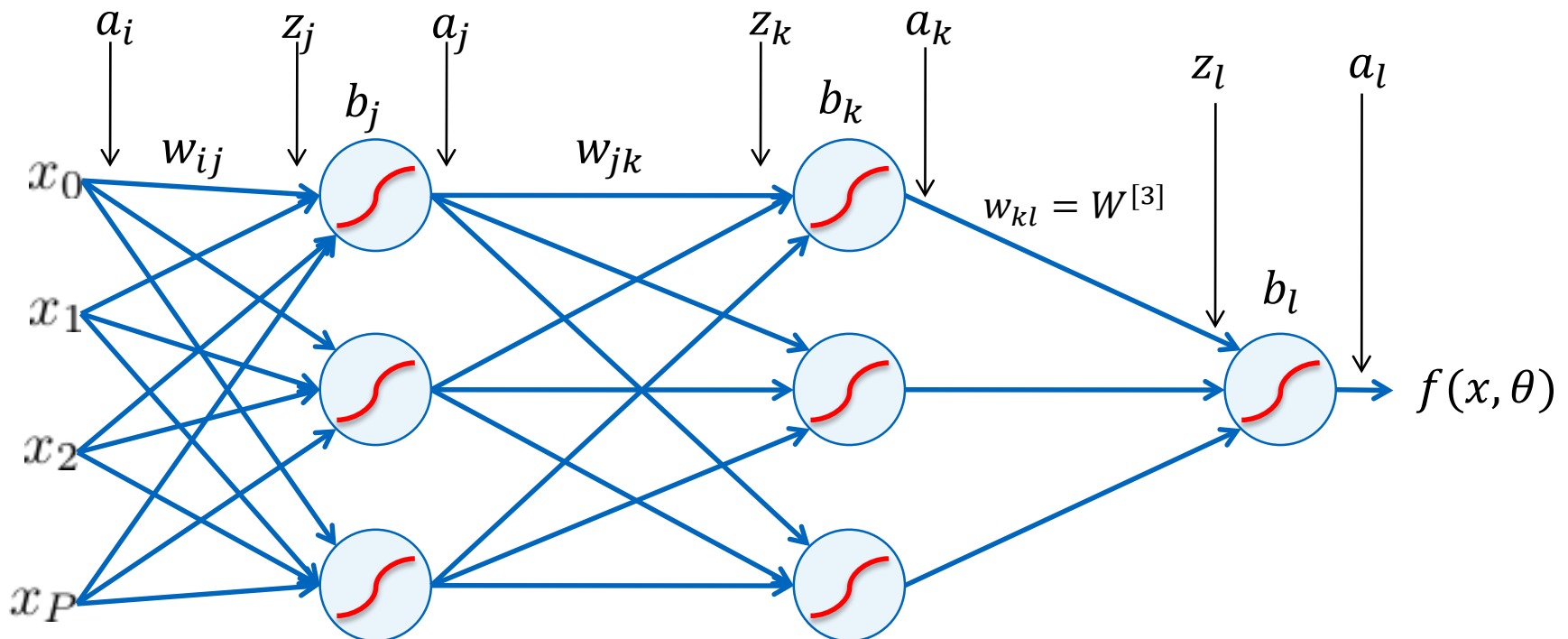
Introduce variables to distinguish the input and output of each node.



Error Backpropagation

Matrix notation: $z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$ $z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$ $z^{[3]} = W^{[3]}a^{[2]} + b^{[3]}$

Scalar notation: $z_j = \sum_i w_{ij}a_i + b_j$ $z_k = \sum_j w_{jk}a_j + b_k$ $z_l = \sum_k w_{kl}a_k + b_l$
 $a_j = g(z_j)$ $a_k = g(z_k)$ $a_l = g(z_l)$

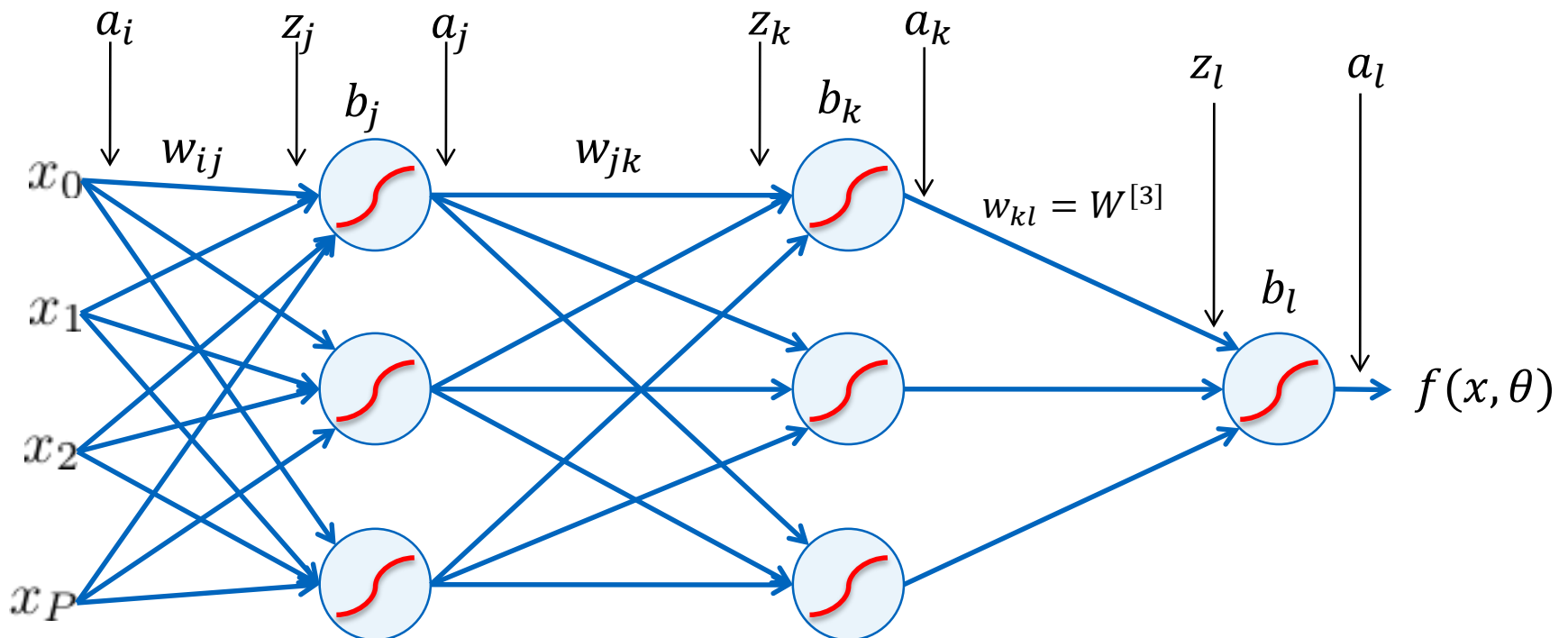


Error Backpropagation

Training: Take the gradient of the last component and iterate backwards.

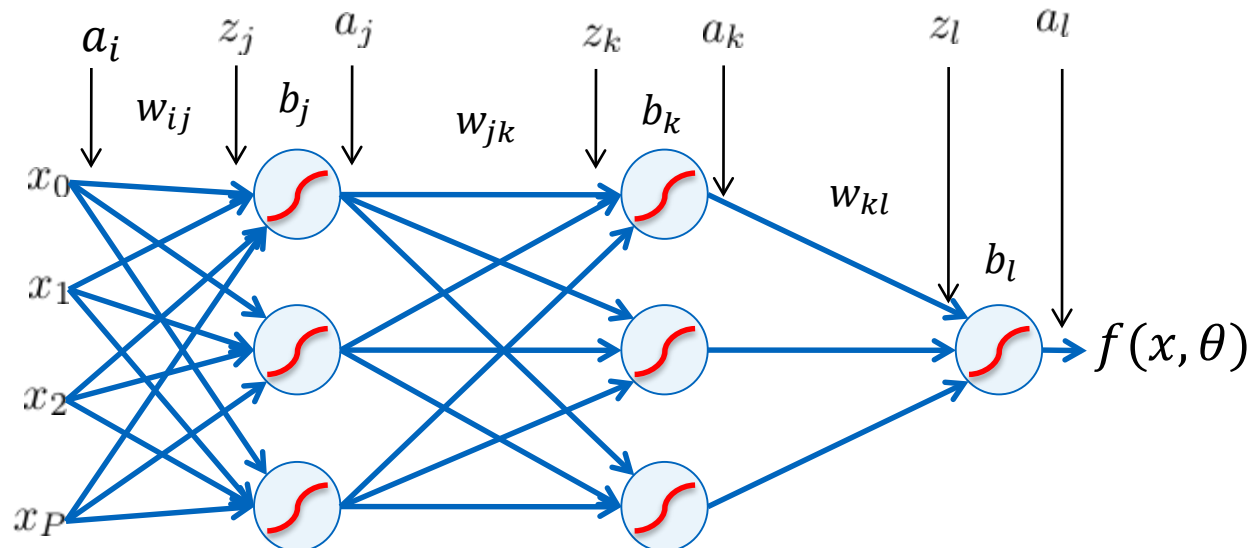
$$z_j = \sum_i w_{ij} a_i + b_j \quad z_k = \sum_j w_{jk} a_j + b_k \quad z_l = \sum_k w_{kl} a_k + b_l$$

$$a_j = g(z_j) \quad a_k = g(z_k) \quad a_l = g(z_l)$$



Error Backpropagation

$$\begin{aligned}
 R(\theta) &= \frac{1}{N} \sum_{n=0}^N L(y_n - f(x_n)) && \text{class label} && \text{Empirical Risk Function} \\
 &= \frac{1}{N} \sum_{n=0}^N \frac{1}{2} (y_n - f(x_n))^2 && \text{introduced for convenience} && \text{NN prediction} \\
 &= \frac{1}{N} \sum_{n=0}^N \frac{1}{2} \left(y_n - g \left(\sum_k w_{kl} g \left(\sum_j w_{jk} g \left(\sum_i w_{ij} x_{n,i} \right) \right) \right) \right)^2 && \text{(ignoring the bias for brevity)}
 \end{aligned}$$

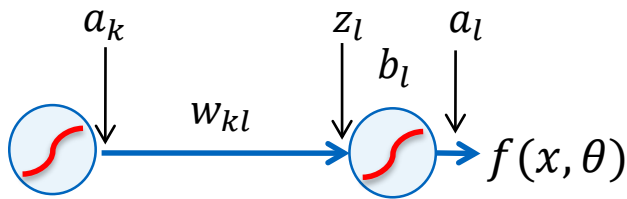


Error Backpropagation

Derive partial derivative wrt. all weights and biases:

$$\nabla R = \begin{pmatrix} \frac{\partial R}{\partial w_{ij}} \\ \frac{\partial R}{\partial b_j} \\ \vdots \\ \frac{\partial R}{\partial w_{kl}} \end{pmatrix}$$

Look at the last layer weights w_{kl} of a simplified network.



$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \frac{\partial L_n}{\partial a_{ln}} \frac{\partial a_{ln}}{\partial z_{ln}} \frac{\partial z_{ln}}{\partial w_{kln}}$$

avg. over training instances

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n [-(y_n - a_{ln}) g'(z_{ln})] a_{kn}$$

$$L_n = \frac{1}{2} (y_n - g(w_{kln} a_{kn} + b_{ln}))^2$$

$$L_n = \frac{1}{2} (y_n - a_{ln})^2$$

$$a_{ln} = g(z_{ln})$$

$$z_{ln} = w_{kln} a_{kn} + b_{ln}$$

$$\frac{\partial L_n}{\partial a_{ln}} = -(y_n - a_{ln})$$

$$\frac{\partial a_{ln}}{\partial z_{ln}} = g'(z_{ln})$$

$$\frac{\partial z_{ln}}{\partial w_{kln}} = a_{kn}$$

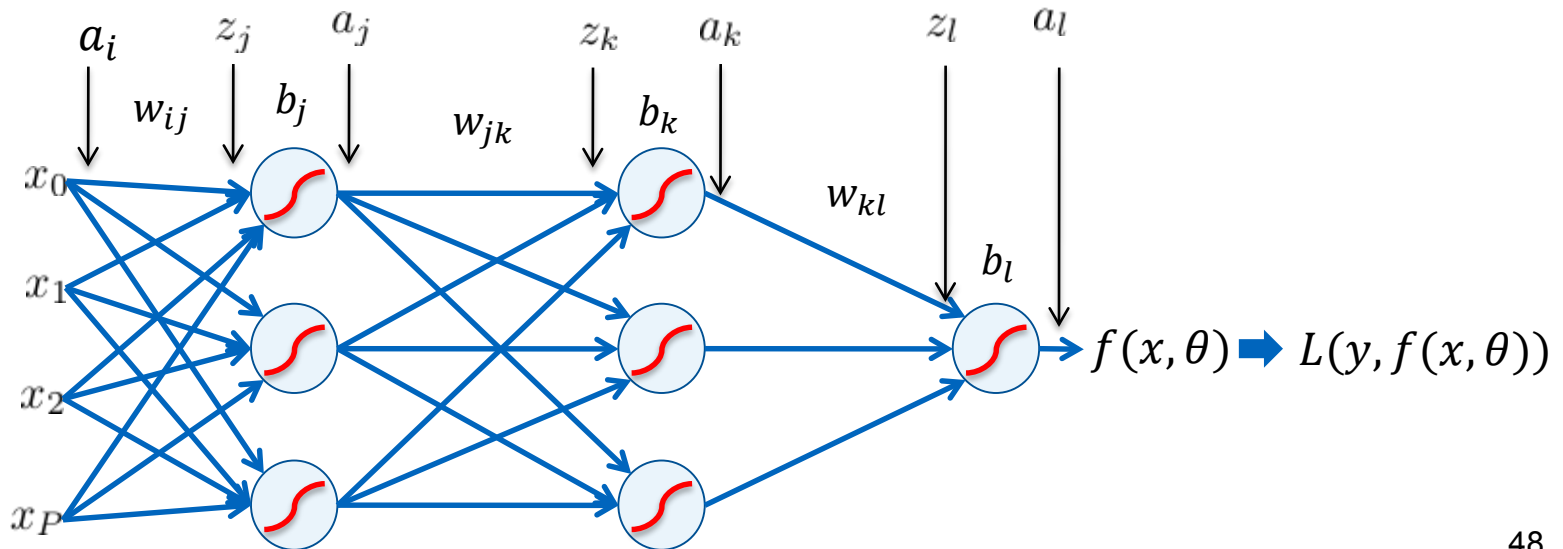
Error Backpropagation

Repeat for all previous layers:

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n [-(y_n - z_{ln}) g'(z_{ln})] a_{kn} = \frac{1}{N} \sum_n \delta_{ln} a_{kn}$$

$$\frac{\partial R}{\partial w_{jk}} = \frac{1}{N} \sum_n [\delta_{ln} w_{kl} g'(z_{kn})] a_{jn} = \frac{1}{N} \sum_n \delta_{kn} a_{jn}$$

$$\frac{\partial R}{\partial w_{ij}} = \frac{1}{N} \sum_n [\delta_{kn} w_{jk} g'(z_{jn})] a_{in} = \frac{1}{N} \sum_n \delta_{jn} a_{in}$$



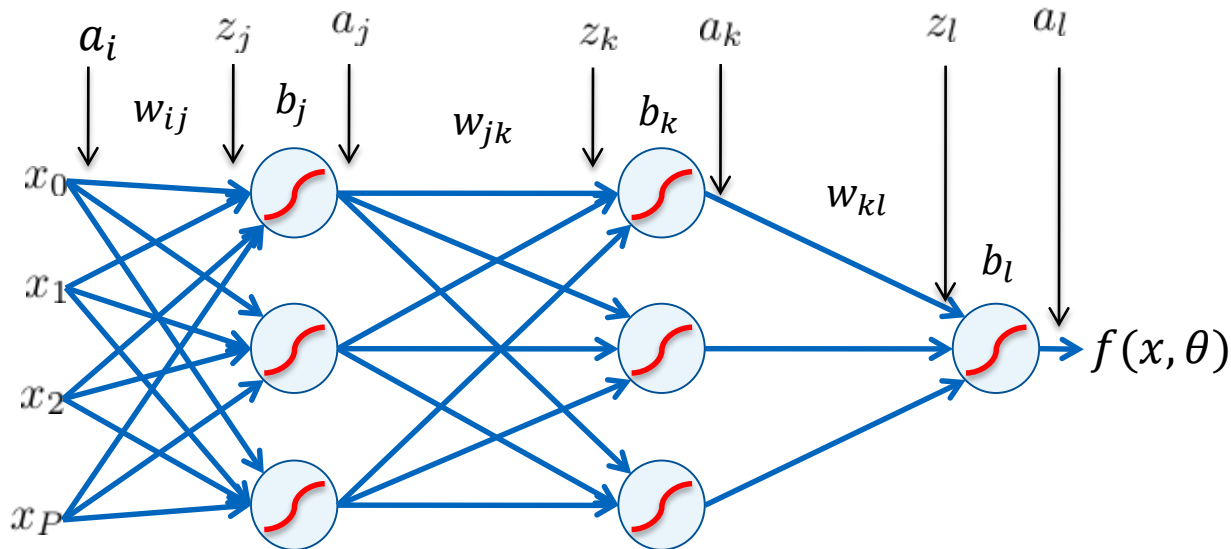
Gradient Descent

Now that we have well defined gradients for each parameter (weight and bias).

Update using gradient descent:

- Add up the changes for particular weight for a batch.
- Multiply by the learning rate.
- Subtract the outcome from the current value of the weight.
- Repeat on previous layers and propagate error backwards.
- Apply the same computation for biases.
- Until $|\theta^{t+1} - \theta^t| < \epsilon$

$$\begin{aligned} w_{ij}^{t+1} &= w_{ij}^t - \alpha \frac{\partial R}{\partial w_{ij}} \\ w_{jk}^{t+1} &= w_{jk}^t - \alpha \frac{\partial R}{\partial w_{jk}} \\ w_{kl}^{t+1} &= w_{kl}^t - \alpha \frac{\partial R}{\partial w_{kl}} \end{aligned}$$



Summary

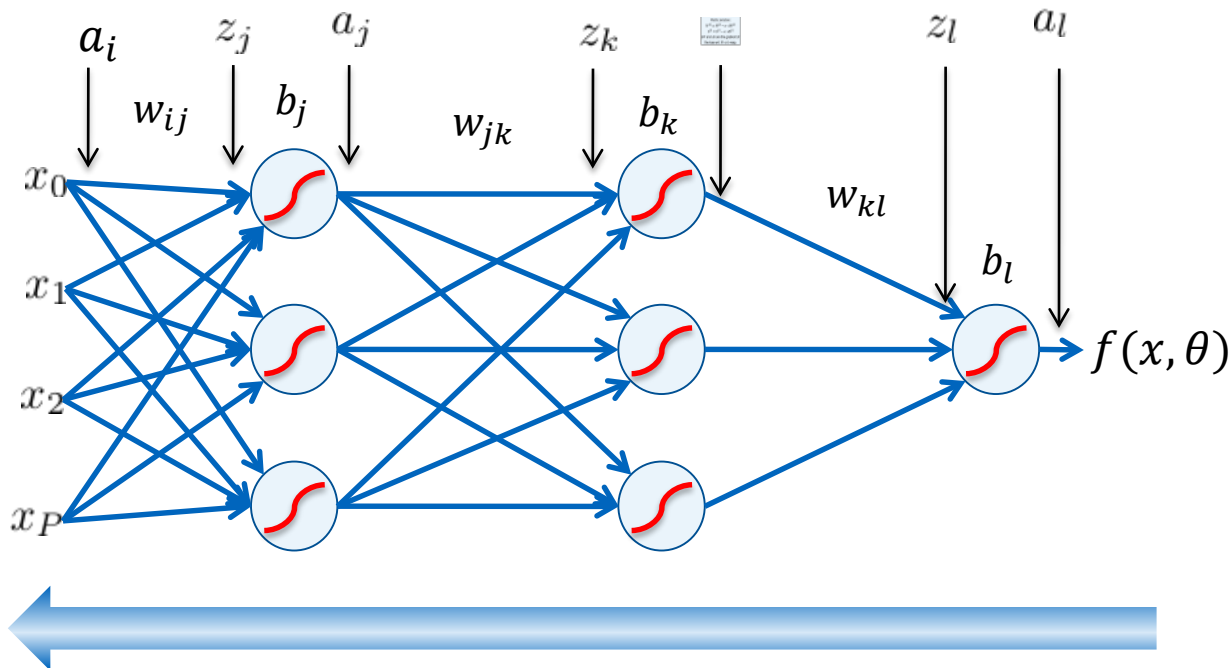
- Error backprop unravels the multivariate chain rule and solves the gradient for each partial component separately.
- The target values for each layer come from the next layer.
- This feeds the errors back along the network.

Matrix notation:

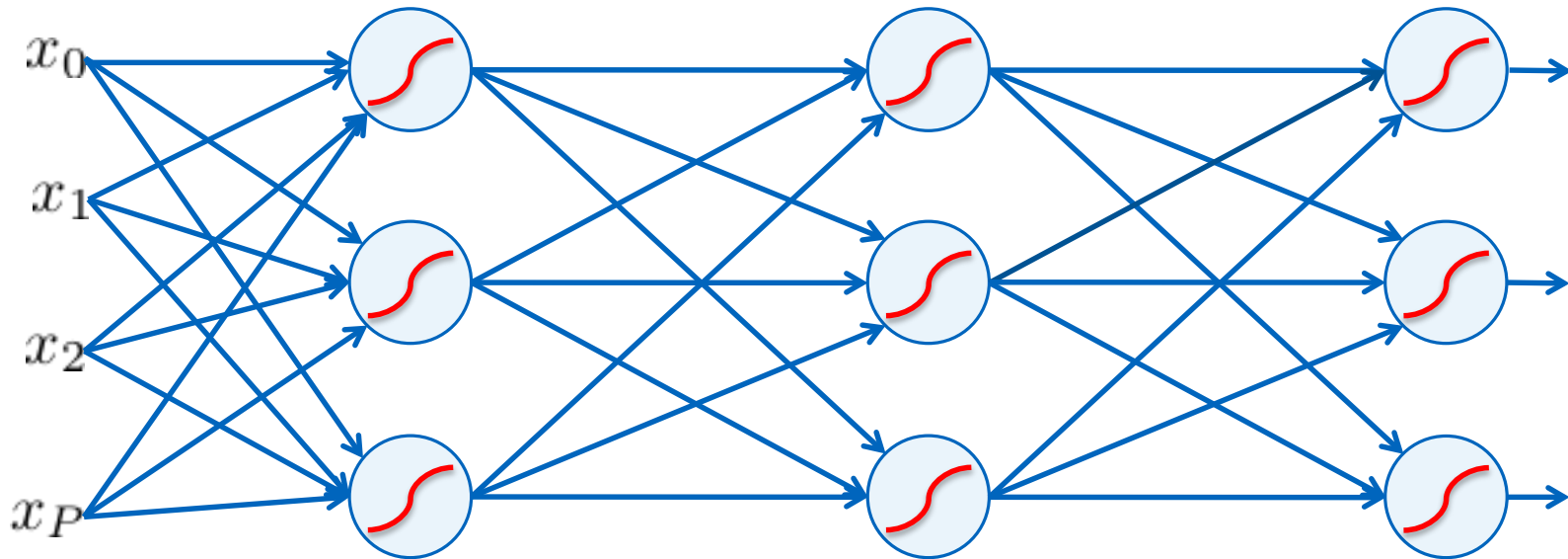
$$\underline{W^{[l]} := W^{[l]} - \alpha \cdot dW^{[l]}}$$

$$\underline{b^{[l]} := b^{[l]} - \alpha \cdot db^{[l]}}$$

dW and db are the gradient of the loss wrt. W or b resp.



Multiple Outputs



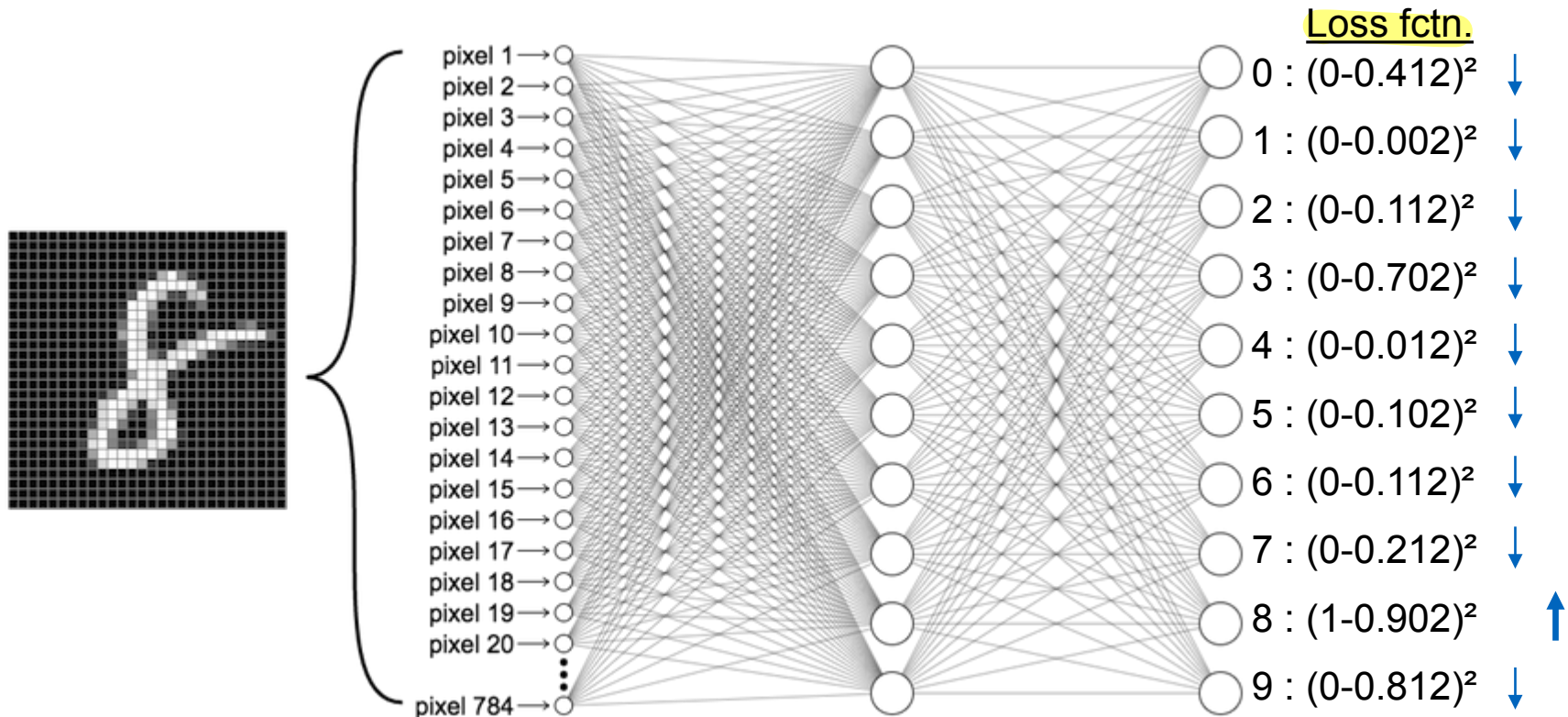
Used for N-way classification.

Each node in the output layer corresponds to a different class.

No guarantee that the sum of the output vector will equal 1.

Example with Multiple Outputs

Backpropagation on the following example adapts the weights and biases for this example such that the **output value for 8 is increased** and that of the **other outputs is decreased**.



```
import torch
import torch.nn as nn
import torch.optim as optim
```

input data X and Y to be defined ...

Define the neural network architecture

```
class SimpleClassifier(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleClassifier, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)
        self.sigmoid = nn.Sigmoid()
```

```
def forward(self, x):
    x = self.fc1(x)
    x = self.relu(x)
    x = self.fc2(x)
    x = self.sigmoid(x)
    return x
```

forward pass

Instantiate the model, loss function, and optimizer

```
input_size = 2 # Number of features in the input data
hidden_size = 10 # Number of neurons in the hidden layer
output_size = 1 # Number of output neurons (binary
classification)
model = SimpleClassifier(input_size, hidden_size, output_size)
```

```
criterion = nn.BCELoss() # Binary Cross Entropy Loss
optimizer = optim.SGD(model.parameters(), lr=0.01)
# Stochastic Gradient Descent
```

Training loop

```
epochs = 1000
```

```
for epoch in range(epochs):
```

```
    # Forward pass
```

```
    y_pred = model(X)
```

calls forward

```
    # Compute the loss
```

```
    loss = criterion(y_pred.squeeze(), y)
```

```
    # Backward pass and optimization
```

```
    optimizer.zero_grad()
```

```
    loss.backward()
```

```
    optimizer.step()
```

Backpropagation!

```
    # Print the loss every 100 epochs
```

```
    if (epoch + 1) % 100 == 0:
```

```
        print(f'Epoch [{epoch + 1}/{epochs}], Loss: {loss.item():.4f}')
```

Testing the trained model with new data

```
new_data = torch.rand((5, 2)) # 5 new samples
```

```
predictions = model(new_data)
```

```
print("Predictions:", predictions.squeeze().detach().numpy())
```

```
import torch
import torch.nn as nn
import torch.optim as optim
```

input data X and Y to be defined ...

Define the neural network architecture

```
class SimpleClassifier(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleClassifier, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.sigmoid(x)
        return x
```

Instantiate the model, loss function, and optimizer

```
input_size = 2 # Number of features in the input data
hidden_size = 10 # Number of neurons in the hidden layer
output_size = 1 # Number of output neurons (binary
classification)
model = SimpleClassifier(input_size, hidden_size, output_size)
```

SimpleClassifier is defined as a subclass of **nn.Module**.

The constructor (`__init__`) initializes the neural network architecture. It consists of:

- **nn.Linear(input_size, hidden_size)**: The first **fully connected (linear) layer** with `input_size` input features and `hidden_size` output features. This layer performs a linear transformation on the input data.
- **nn.ReLU()**: The **Rectified Linear Unit (ReLU) activation function** applied element-wise after the first linear layer. ReLU introduces **non-linearity** to the model by setting **negative values to zero**.
- **nn.Linear(hidden_size, output_size)**: The **second fully connected layer** with `hidden_size` input features and `output_size` output features. This layer produces the final output of the model.
- **nn.Sigmoid()**: The **sigmoid activation function** applied to the output of the second linear layer. This function **squashes the output between 0 and 1**, making it **suitable for binary classification tasks**.

The **forward method** defines how the input data `x` flows through the neural network during a **forward pass**. It applies the operations defined in the `__init__` method sequentially. The output of the final layer is returned, representing the model's prediction for the input.

The **model's architecture** is defined with an input size of 2, a hidden layer with 10 neurons, and a single output neuron for binary classification.

```
criterion = nn.BCELoss() # Binary Cross Entropy Loss
optimizer = optim.SGD(model.parameters(), lr=0.01)
# Stochastic Gradient Descent
```

```
# Training loop
```

```
epochs = 1000
```

```
for epoch in range(epochs):
```

```
    # Forward pass
```

```
    y_pred = model(X)
```

```
    # Compute the loss
```

```
    loss = criterion(y_pred.squeeze(), y)
```

```
    # Backward pass and optimization
```

```
    optimizer.zero_grad()
```

```
    loss.backward()
```

```
    optimizer.step()
```

```
# Print the loss every 100 epochs
```

```
if (epoch + 1) % 100 == 0:
```

```
    print(f'Epoch [{epoch + 1}/{epochs}], Loss: {loss.item():.4f}')
```

```
# Testing the trained model with new data
```

```
new_data = torch.rand((5, 2)) # 5 new samples
```

```
predictions = model(new_data)
```

```
print("Predictions:", predictions.squeeze().detach().numpy())
```

The **loss function** (BCELoss for Binary Cross Entropy Loss), and **optimizer** (SGD for Stochastic Gradient Descent) are instantiated.

The **training loop** runs for 1000 epochs. In each epoch:

Forward pass: The model predicts the output (y_pred) for the input data X.

Compute the loss:

The Binary Cross Entropy Loss between the predicted output and the ground truth labels y is computed.

Backward pass and optimization: Gradients are computed with respect to the loss, and the optimizer updates the model's parameters to minimize the loss.

The loss is printed every 100 epochs to monitor the training progress.

Finally, the trained model is tested with new randomly generated data (new_data).

The predictions for the new data are printed.

What can we do to speed up batched gradient descent?



Speeding Up Gradient Descent

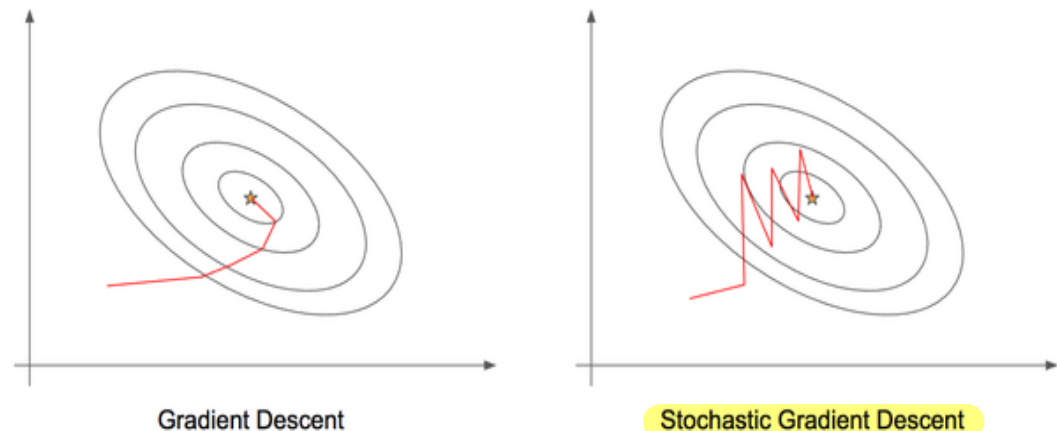
An **epoch** in (standard) gradient descent is one pass through the training set, with an adjustment via backpropagation to the network weights.

Remember that the **loss function is large** as it contains every single element of the entire data set with N samples. Then the **gradient needs to be computed for each iteration of gradient descent many times.**

$$\nabla_{\theta} R = \frac{1}{2N} \sum_n 2(y_n - g(\theta^T x_n))(-1)g'(\theta^T x_n)x_n = 0$$

Instead compute the loss and the gradient on a **small random sample** (~1-1000 samples) or batch size in **mini-batch GD** or even on instances using **Stochastic Gradient Descent**.

Result: fast, but noisy gradient.



Reminder: Mini-Batch Gradient Descent

In mini-batch gradient descent, the model makes a forward pass through all instances in a mini-batch (e.g. a few hundred instances) and produces a prediction for each one.

Then the average errors for all instances in the mini-batch is backpropagated through the network, which updates the model's parameters. A smaller mini-batch size can provide more noisy gradients, but is faster to compute.

Example: if you have 1000 training examples, and your batch size is 500, then it will take 2 iterations to complete 1 epoch.

Mini-batch GD is typically more efficient than standard GD and it can be parallelized.

The batch size is an important hyperparameter. It is often preferred in practice.

Momentum uses an exponential averaging of gradients to make sudden changes in direction less likely. It can help in SGD.

Reminder: Regularization

We can use L2 or L1 regularization with adding a regularization term to the loss function.

Loss function $L(y, \hat{y}, \theta) = \sum_{i=1} (\hat{y}_i - y_i)^2 + \lambda R(\theta)$

$R(\theta) = \sum_i \theta_i^2$ for L2 regularization or $R(\theta) = \sum_i |\theta_i|$ for L1 regularization.

Suppose there are three features. A regularized classifier might not take all features into account.

$$\theta_1 = [0.75, 0, 0] \quad \sum_n |\theta| = 0.75 \text{ (min)}$$

$$\theta_2 = [0.25, 0.5, 0.25] \quad \sum_n |\theta| = 1$$

Alternative Optimizers

Adagrad adapts the learning rate for each parameter separately, based on historical gradient information.

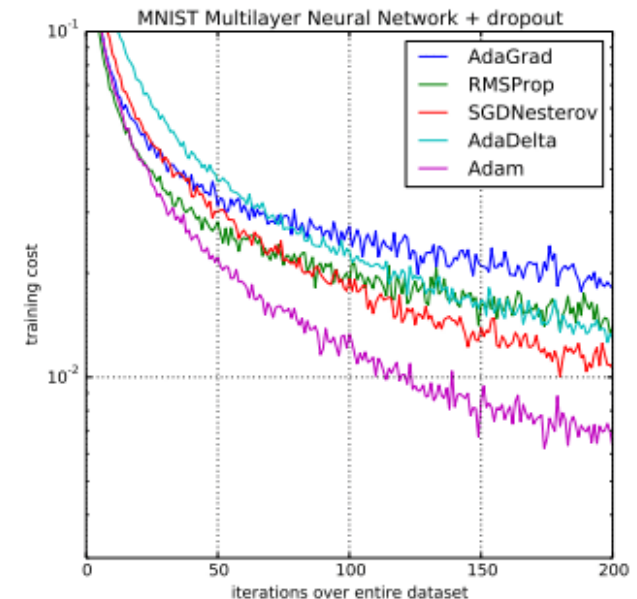
$$\theta^{t+1} = \theta^t - \frac{\alpha}{G_t} \nabla_{\theta} R^t$$

Divides the learning rate by an exponentially decaying average of previous squared gradients G_t .

RMSProp also adapts the learning rates for each parameter individually, but it uses a moving average of the historical gradient information, rather than the accumulation of the historical gradient information.

Adam is similar to RMSProp, but uses two moving averages and a bias correction term to update the parameters. *Adam is widely used today.*

AdaMax, Nadam, AMSGrad, etc.



Source: Kingma, Diederik P., and Jimmy Ba.
 "Adam: A method for stochastic optimization."
 arXiv preprint arXiv:1412.6980 (2014).

Hyperparameters

- Number of hidden layers
- Number of nodes in each hidden layer
- Activation function
 - Sigmoid suffers from vanishing gradient, ReLU is now widespread
- Learning rate and momentum
- Iterations and desired error level
- Batch size in mini-batch GD
- Other optimization algorithms than SGD such as RMSProp, Adam, etc.
- Regularization (aka. weight decay) in each layer (as in ridge regression) to limit the influence of irrelevant connections with low weight on the network's predictions

$$R(\theta) = \arg \min_{\theta} \left[\frac{1}{N} \sum_n L(y_n, f(x_n, \theta)) + \lambda R(\theta) \right]$$

Setting the Number of Nodes in the Hidden Layer

The number of nodes in the hidden layer affects generality and convergence.

If too few hidden nodes: convergence may fail.

Few but not too few nodes: possibly slow convergence but good generalization.

Too many hidden nodes (breadth): Rapid convergence, but “overfitting” happens.

Interpretation of Hidden Layers

What are the hidden layers doing?!

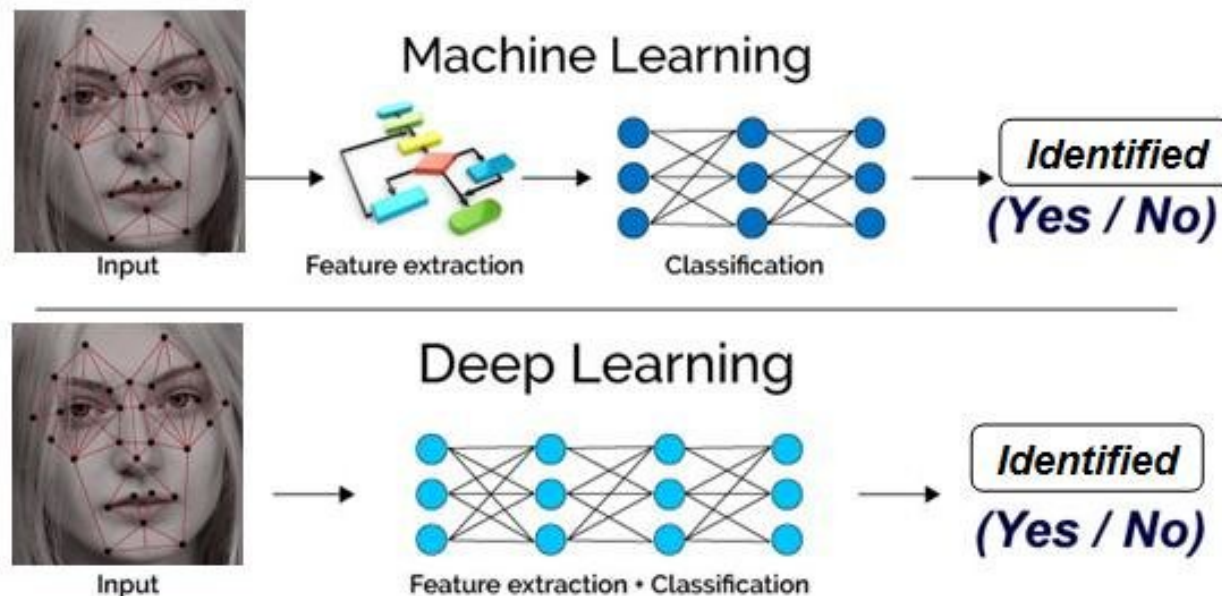
Feature extraction

The non-linearities in the feature extraction can make interpretation of the hidden layers very difficult.

This leads to Neural Networks being treated as black boxes.

Deep Learning

- Usually having **two or more** hidden layers counts as deep.
- Typically trained on GPUs.
- Seen as an end-to-end framework (no feature extraction steps necessary).



- **Feature engineering remains valuable**, and the choice between manual feature engineering and end-to-end learning depends on the task at hand and the available data.

Universal Approximation Theorem

A feed-forward neural network with a single hidden layer and continuous non-linear activation function can approximate any continuous function with arbitrary precision.*

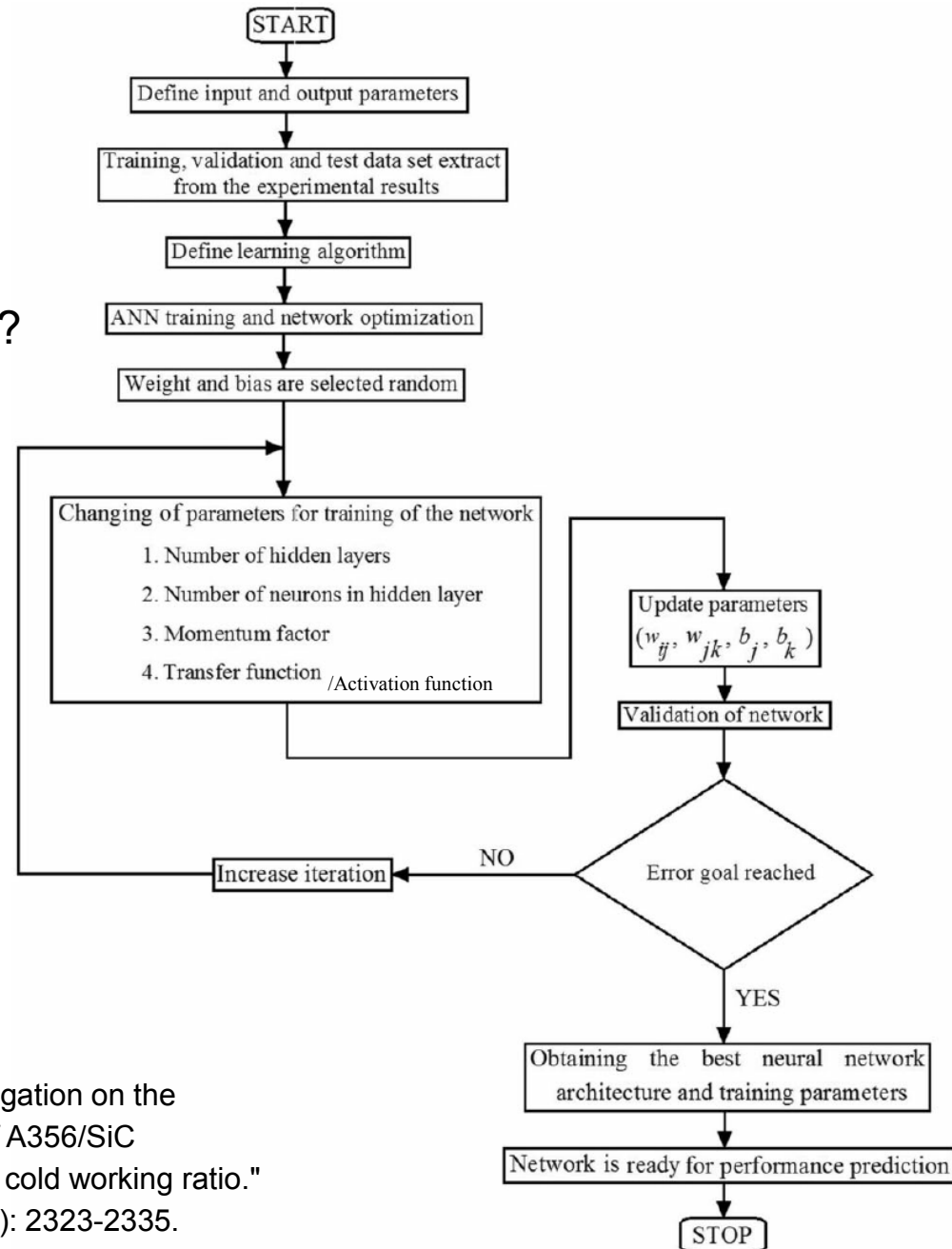
- | | | |
|---|---|--|
| <ul style="list-style-type: none">1. Neural Network2. One hidden layer3. Non-linear activation function | } | <p>Approximate any function
with any precision</p> |
|---|---|--|

Deep Neural Networks $>_{\text{in practice}}$ Wide Neural Networks

*Hornik, Kurt; Tinchcombe, Maxwell; White, Halbert (1989). *Multilayer Feedforward Networks are Universal Approximators*. *Neural Networks*. 2. pp. 359–366.

A Process Model

So, what do I have to do now?

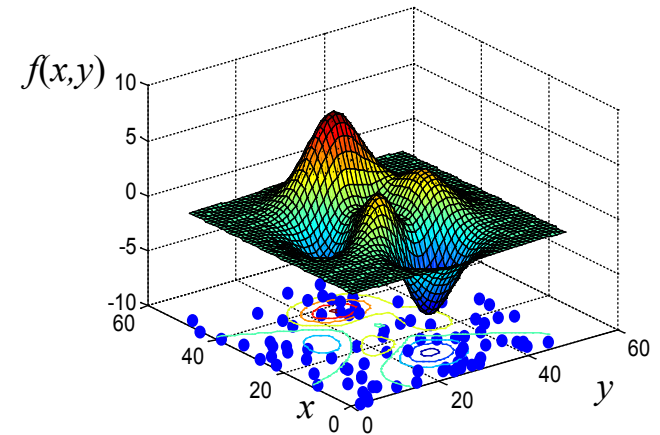
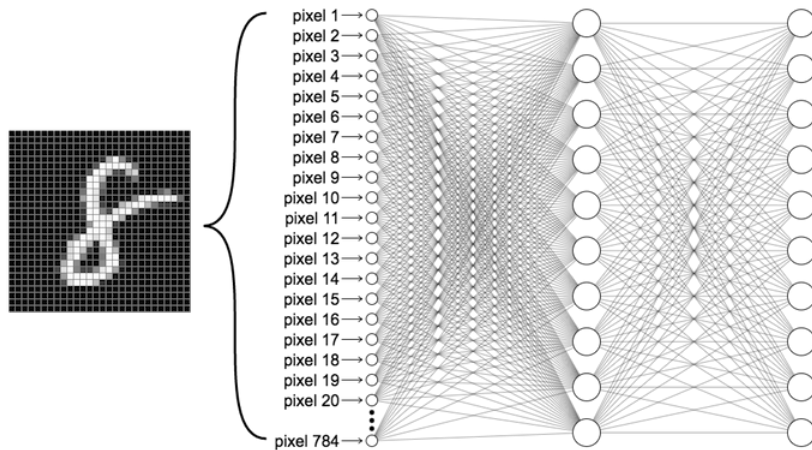


Tuntas, Remzi, and Burak Dikici. "An investigation on the aging responses and corrosion behaviour of A356/SiC composites by neural network: The effect of cold working ratio." *Journal of Composite Materials* 50.17 (2016): 2323-2335.

Machine Learning and Optimization

Empirical risk minimization:

Fitting the parameters of the model („training“) = optimization



$$R(\theta) = \frac{1}{N} \sum_n L(y_n, g(\theta, x_n)) + R(x_n)$$

$N = \#$ examples

$y_n =$ labels

$\theta \in \mathbb{R}^d =$ features

$R(x_n) =$ regularization

Global optimization is NP-hard, convex optimization is not!

Lots of Developments! Here are a Few ...

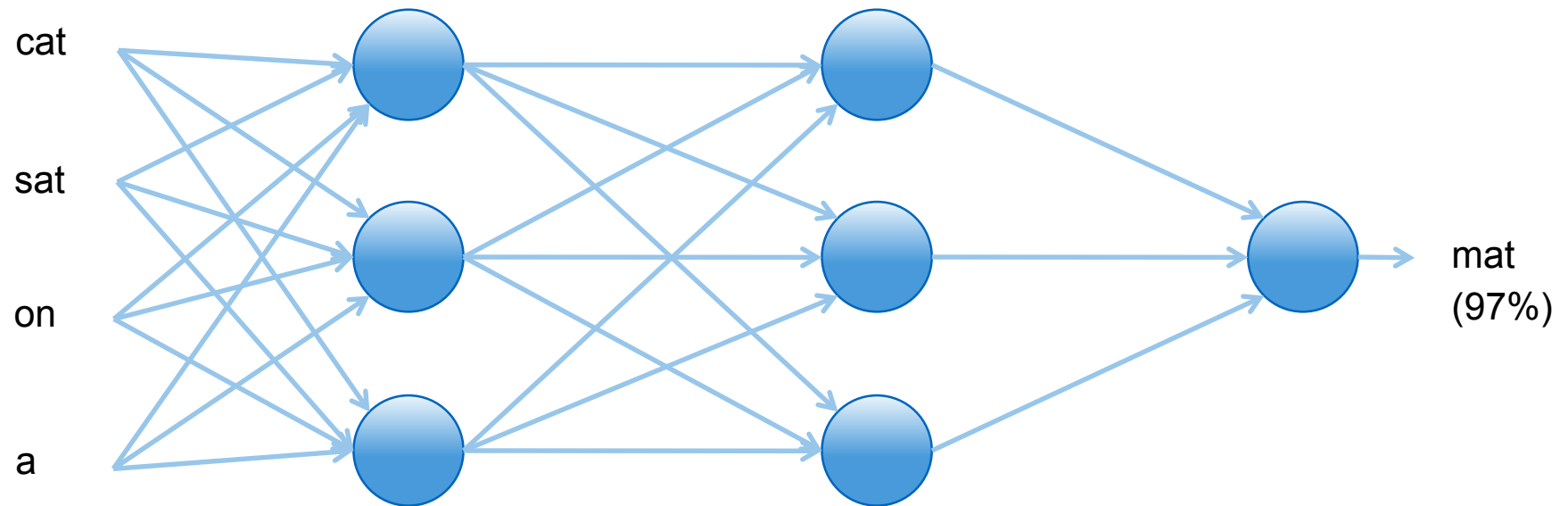
GANs: A GAN is a generative model that produces new, synthetic instances of data that can pass for real data. It does so by using two sub-networks: Generator and Discriminator. For a given input, the generator produces synthetic data that closely resembles the input, and the discriminator distinguishes them.

Diffusion Models: Techniques based on non-equilibrium thermodynamics nowadays widely used for image denoising, image generation, etc. DALL-E by OpenAI is an example.

Autoencoders are a type of artificial neural network used to learn efficient codings of unlabeled data (unsupervised learning). An autoencoder learns two functions: an encoding function that transforms the input data, and a decoding function that recreates the input data from the encoded representation. The autoencoder learns an efficient representation (encoding) for a set of data, typically for dimensionality reduction.

Transformers: The Transformer is a specific neural network architecture introduced in the paper "Attention is All You Need" by Vaswani et al. in 2017. It was designed to handle sequential data, particularly well-suited for natural language processing tasks. Transformers are frequently used as architecture of **Large Language Models** such as GPT or BERT.

Large Language Models



Large Language Models (LLMs)

At their most basic LLMs are statistical pattern-recognition and prediction systems

LLMs output the next likely word (“token”) in a sentence (“sequence”)

- token: unit of text e.g. word, character. 1 word ~ 0.75 token
- sequence: context - section (“window”) of text e.g. sentence, paragraph, book
- input into chatGPT is 4096 tokens; Claude 2 is 100K tokens

The likelihood of the next word appearing is determined by

- the context in which the words are seen in a larger body of text (“corpus”) and
- the input to the chat (the prompt)

Commercial LLMs are huge:

- **Llama** (by Meta) is an open 70,000,000,000 parameter language model. (~140 GB file)
 - Model inference can be done on your laptop
 - Training is VERY expensive
 - ~10TB of text from the Internet, 6,000 GPUs for 12 days, ~1e24 FLOPS, ~\$2 million
- **Google Brain** and **GPT-4** are reported to have >1,000,000,000,000 parameters.

How are LLMs Trained?

Stage 1: **Pretraining** (yearly)

1. Download ~10TB of text.
2. Get a cluster of ~6,000 GPUs
3. Compress the text into a NN (~\$2 million, 12 days)
4. Obtain a base model

Stage 2: **Finetuning** (weekly)

1. Write labeling instructions
2. Hire people and collect 100K high quality ideal Q&A responses
3. Finetune base model on this data (~1 day)
4. Obtain an assistant model.
5. Run evaluations
6. Deploy
7. Monitor and collect misbehaviors
8. Go to step 1 and fix misbehavior via Q&A as training data

<USER>

Can you write a short introduction about the relevance of the term "monopsony" in economics? Please use examples related to potential monopsonies in the labour market and cite relevant research.

<ASSISTANT>

"Monopsony" refers to a market structure where there is only one buyer for a particular good or service. In economics, this term is particularly relevant in the labor market, where a monopsony employer has significant power over the wages and working conditions of their employees. The presence of a monopsony can result in lower wages and reduced employment opportunities for workers, as the employer has little incentive to increase wages or provide better working conditions...

Performance of Larger and Larger LLMs

