Deep Learning Basics

Computer Vision: CS 600.461/661

Deep Learning Basics

Train functions from data to accomplish visual tasks

Topics:

(1) Deep Neural Network Basics

(2) Fitting DNNs to data

(3) Measures of fit and good training practices

(4) Convolutional Networks

What Is “Deep Learning”?

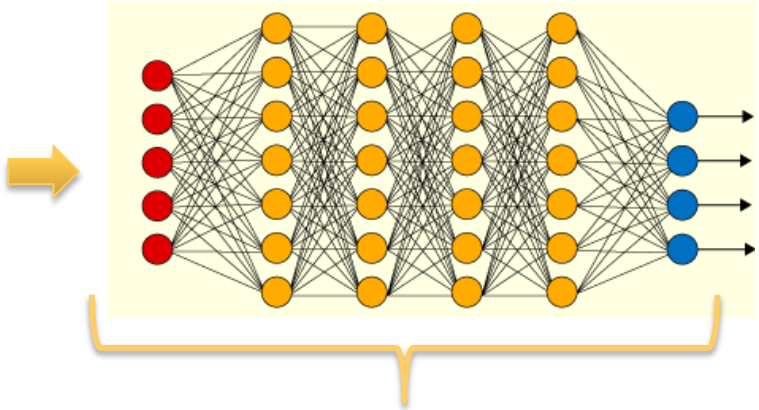
Most Common: Supervised Learning

F()

“Brown Bear”

What Is “Deep Learning”?

Most Common: Supervised Learning



“Brown Bear”

$F(\text{  };W)$

A few 10’s of millions of parameters that are fit to a very large **labelled** data set including lots of brown bears

The Perceptron

$$f\colon \mathbb{R}^d \to \{1,0\}$$

$$f(x) = \begin{cases} 1 & \text{if } \; w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

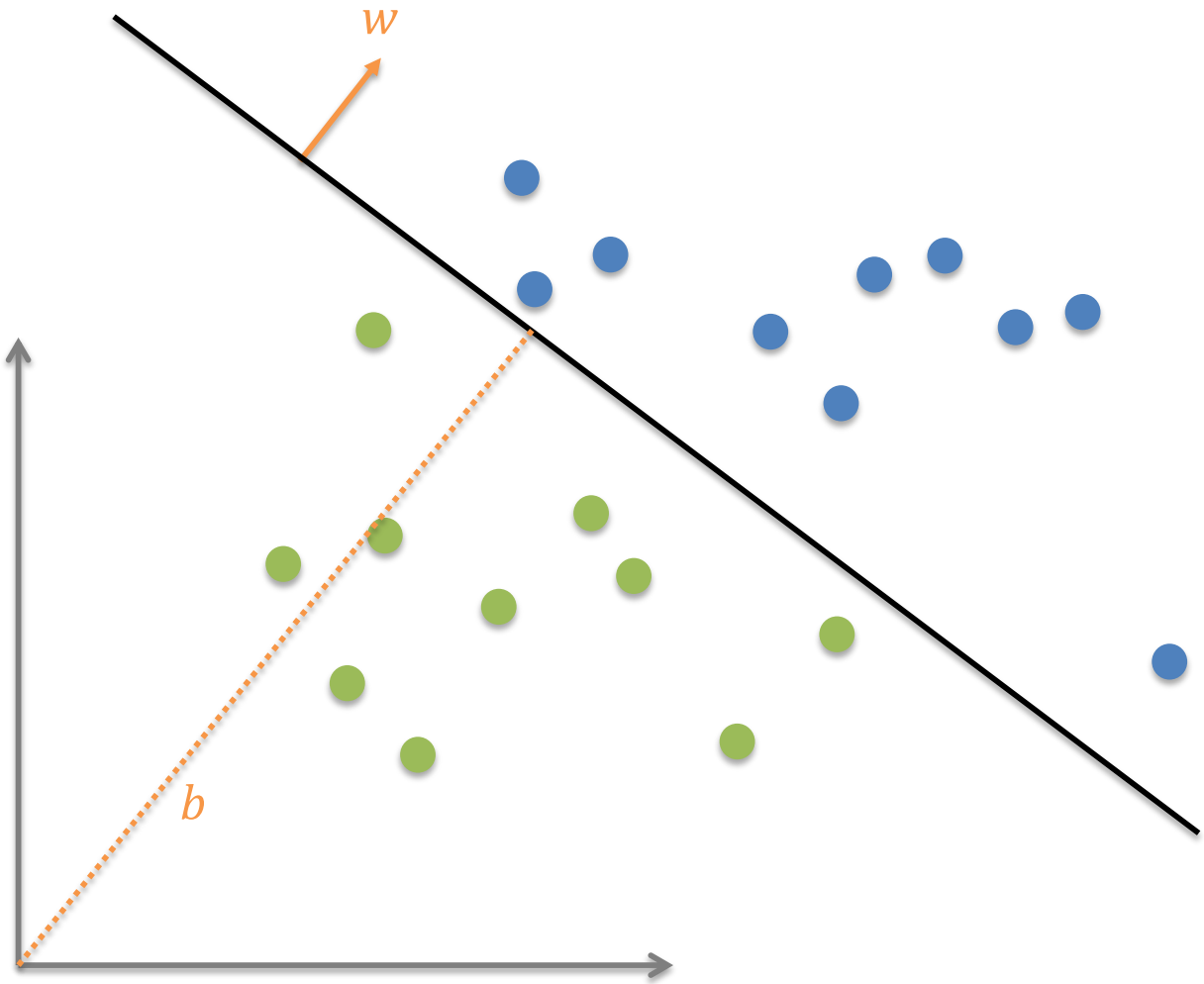
$$x, w \in \mathbb{R}^d \qquad b \in \mathbb{R}$$

Linear Classifier

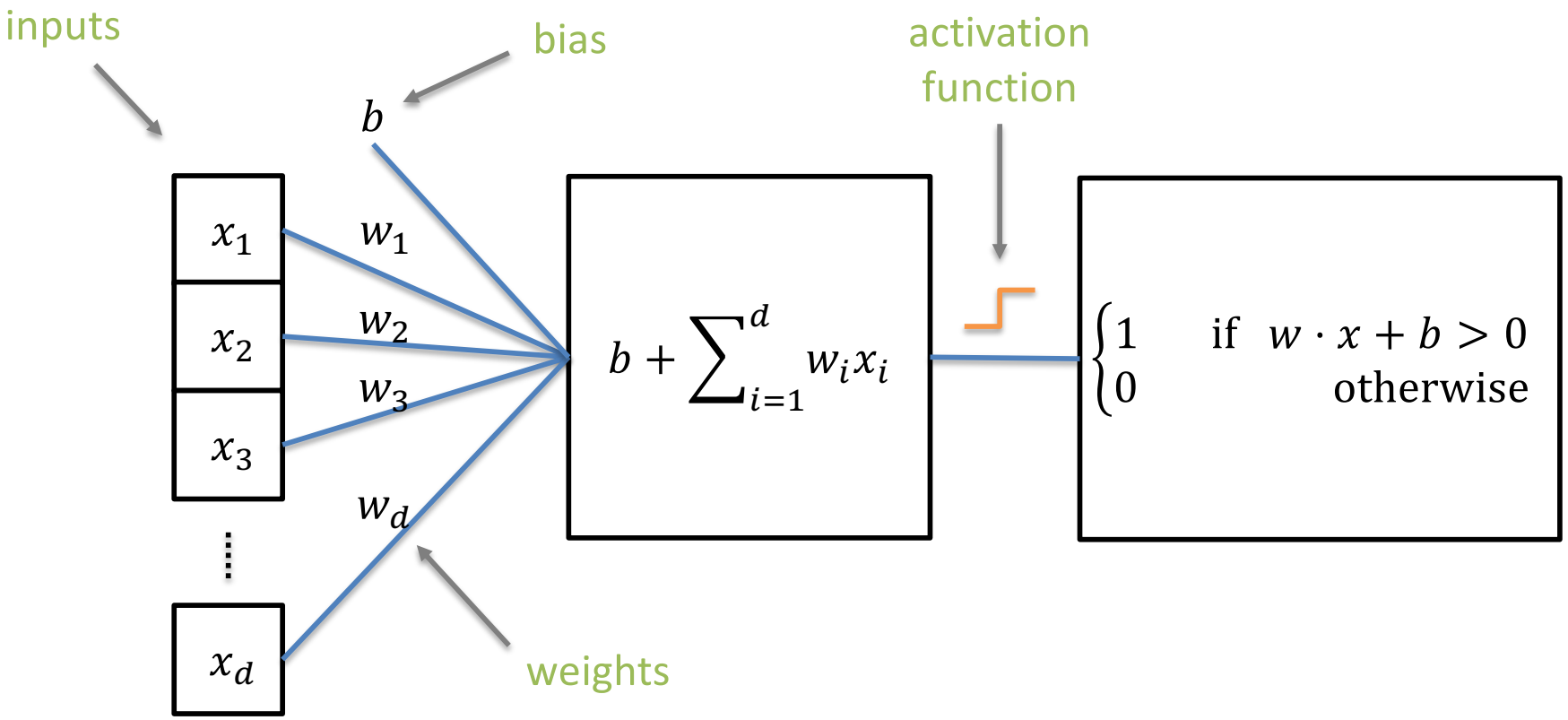
Lead to invention of support vector machines (SVM)

Only works well on linearly separable problems

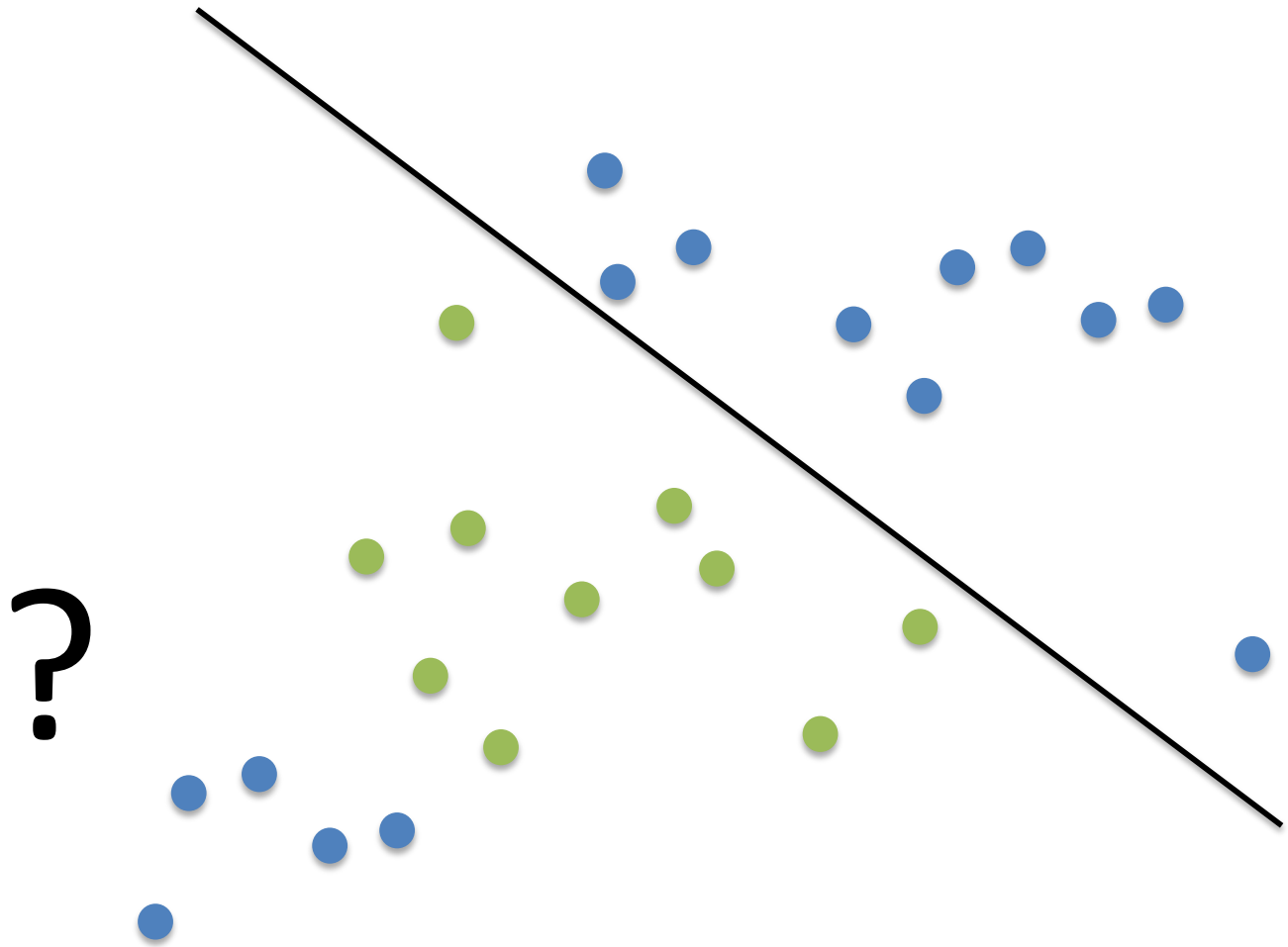
The Perceptron



The Perceptron



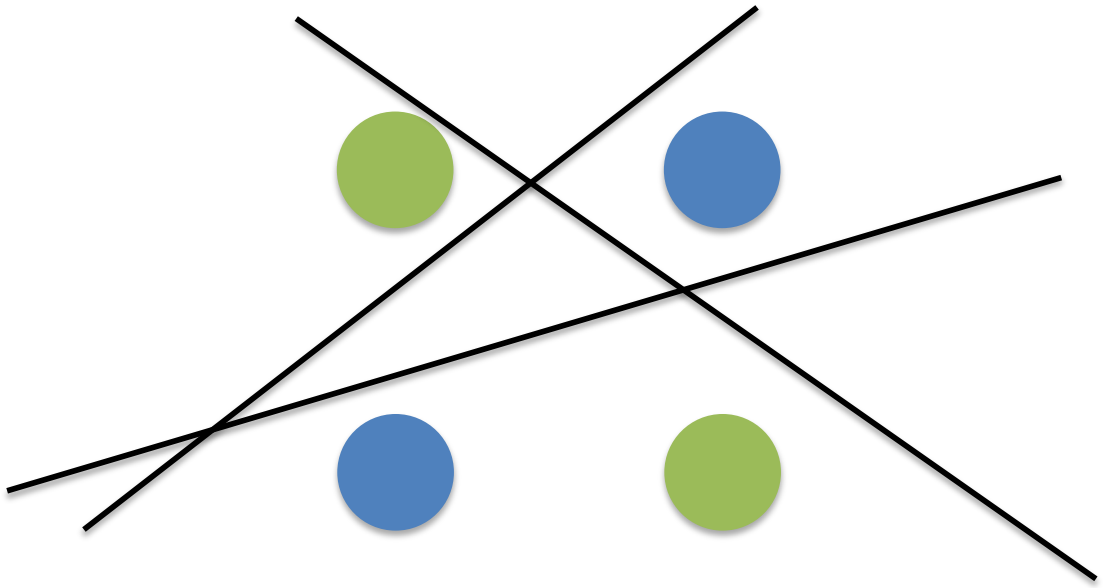
The Perceptron



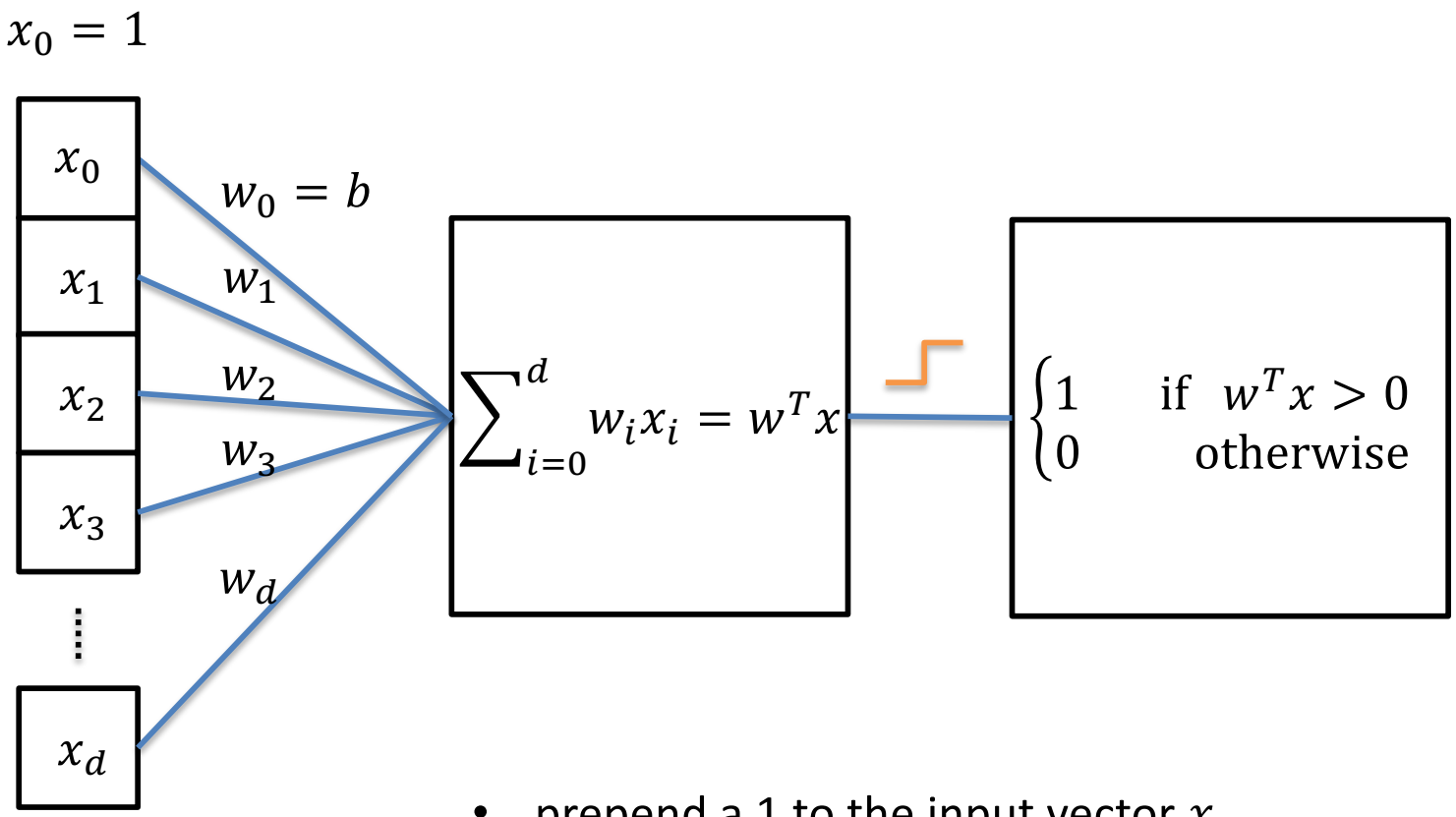
what to do when the problem is not linearly separable?

XOR

Minsky, Marvin, and Seymour Papert. "Perceptrons." (1969): Perceptrons cannot learn the XOR function

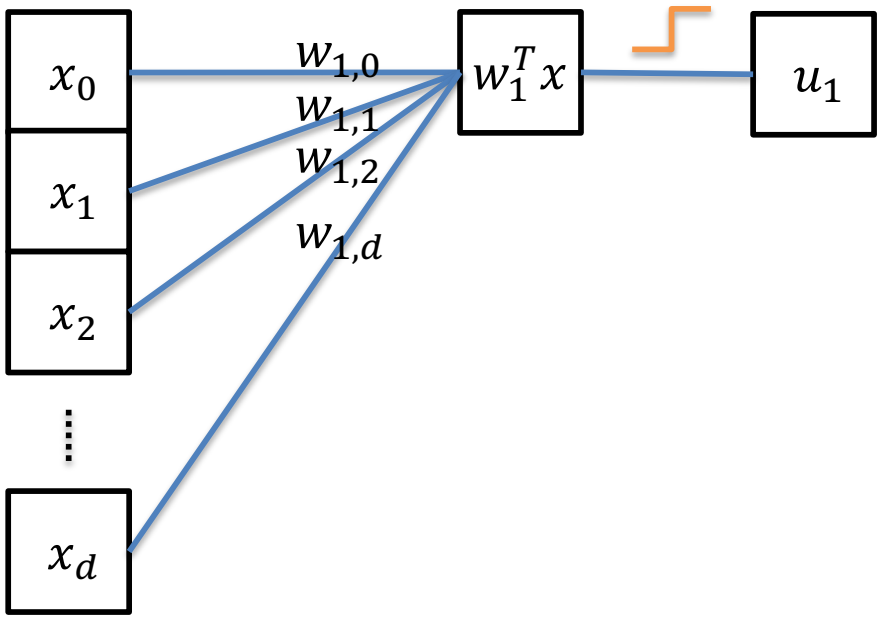


More Layers

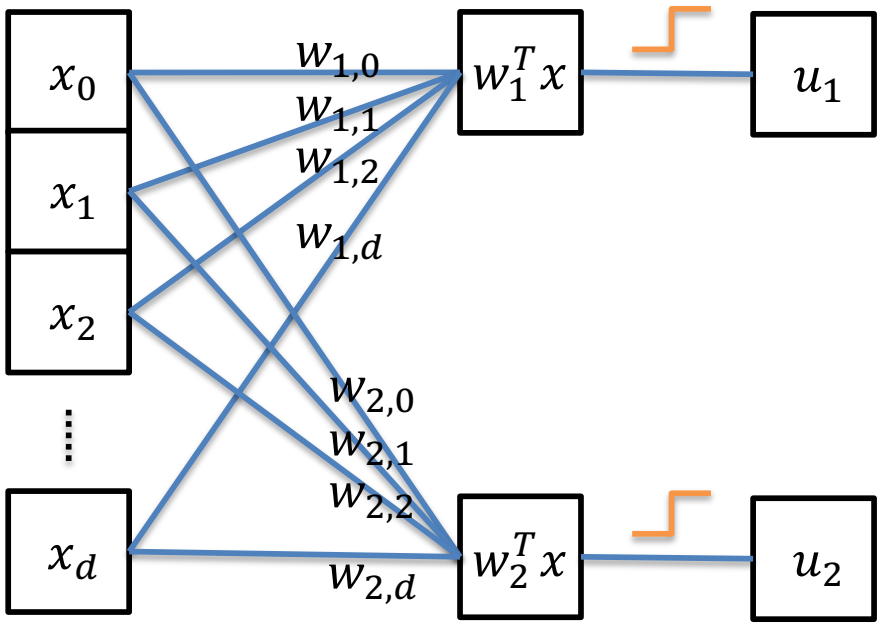


- prepend a 1 to the input vector x
- include the bias into the weights
- write everything as an inner product

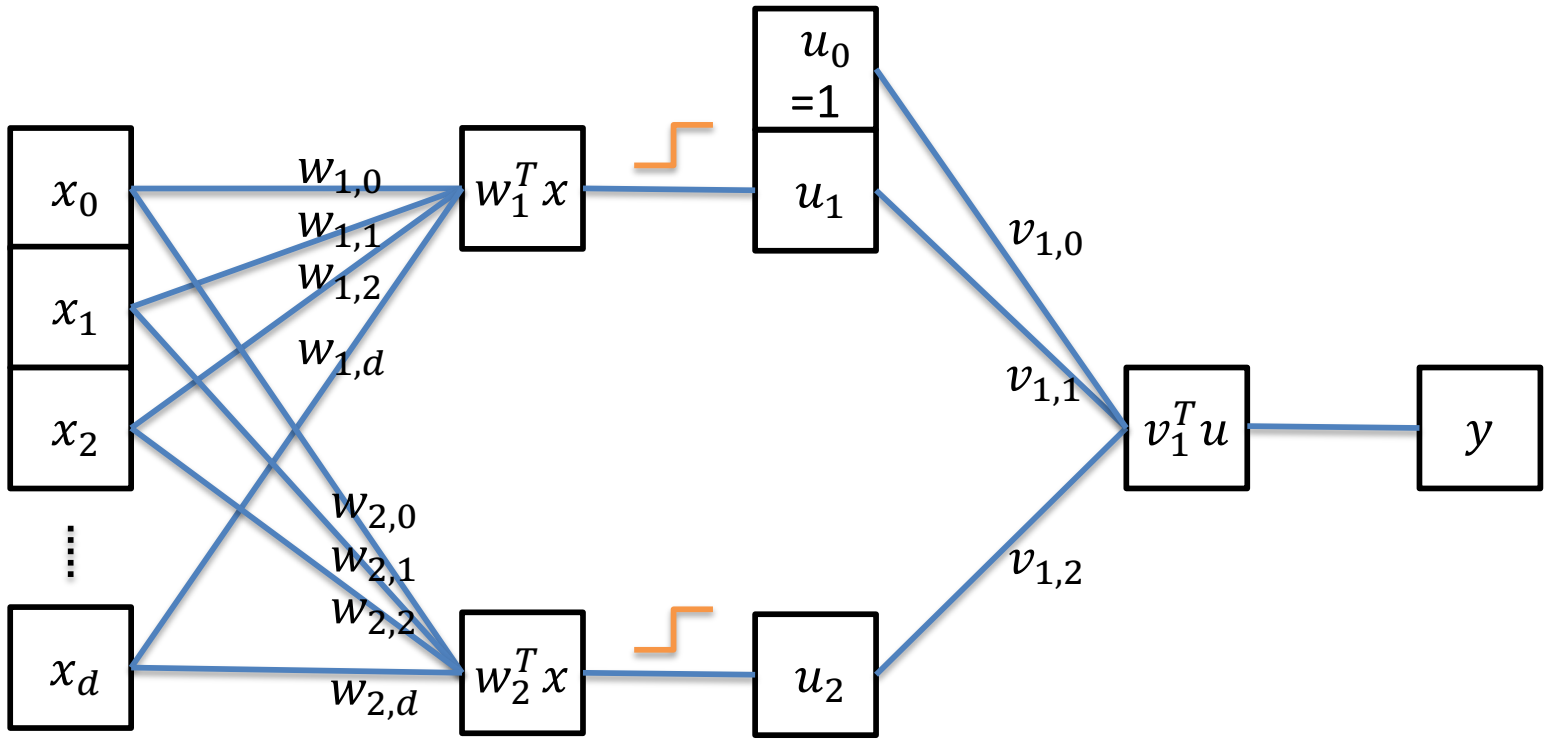
More Layers



More Layers



More Layers

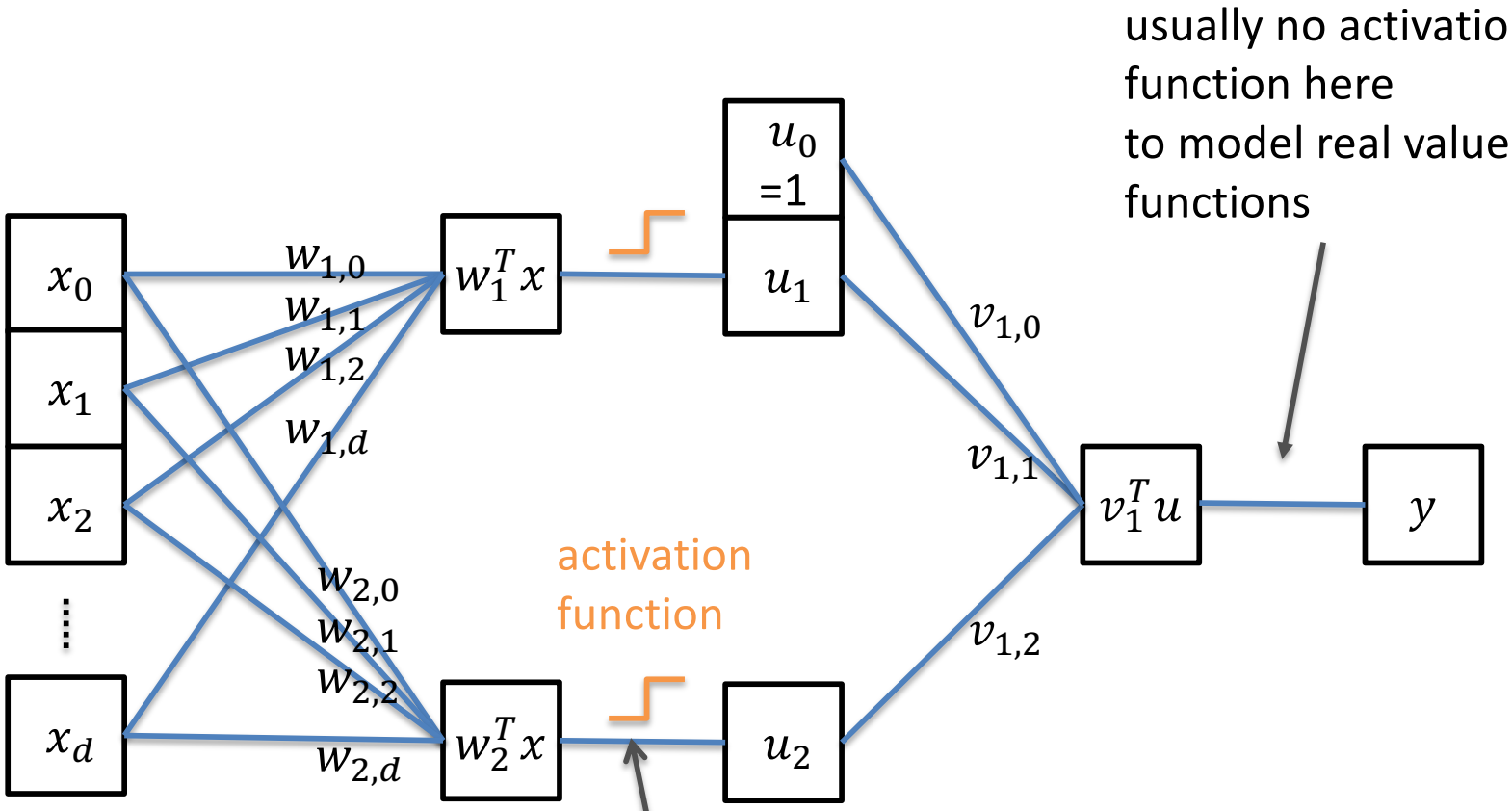


input layer

hidden layer

output layer

More Layers



$$u_j = \sigma\left(\sum_{i=0}^{d} w_{j,i}\, x_i\right) = \sigma(w_j\, x)$$

$$y = \sum_{j=0}^{2} u_{1,j}\, v_j$$

More Layers

$$u_j = \sigma(\sum_{i=0}^{d} w_{j,i}\, x_i) = \sigma(w_j\, x)$$

$$u = \sigma(wx)$$

$$y = uv = v\sigma(wx)$$

Matrix notation greatly simplifies writing down the computations in the layers

Universal Approximation Theorem

If f^* satisfies mild regularity conditions, then for every quality level $\epsilon > 0$, it is possible to choose the parameters of

$$f(x) = \sum_i^N v_i \sigma(w_i^T x + b_i)$$

such that

$$|f(x) - f^*(x)| < \epsilon \quad \forall x,$$

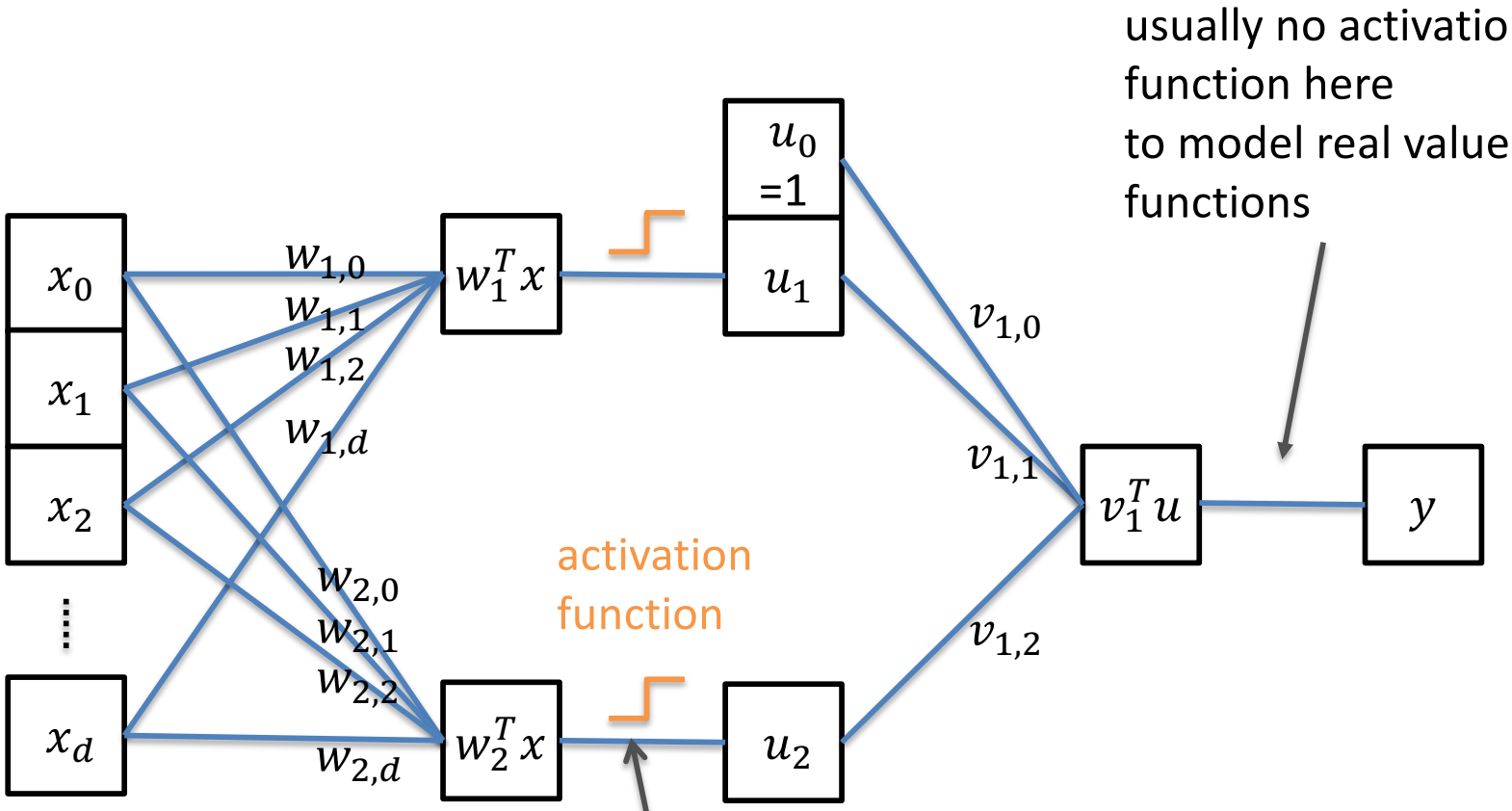
given that the number of parameters N is large enough.

Cybenko., G. (1989) "Approximations by superpositions of sigmoidal functions", Mathematics of Control, Signals, and Systems, 2 (4), 303-314

Gradient-Based Learning

- Specify
  - Model
  - Cost
- Design model and cost so cost is smooth
- Minimize cost using gradient descent or related techniques

The Model: Layered Networks

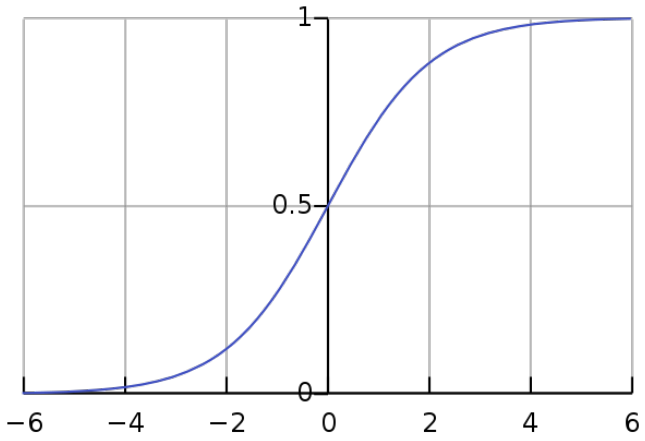


$$u_j = \sigma\left(\sum_{i=0}^{d} w_{j,i}\, x_i\right) = \sigma(w_j\, x)$$

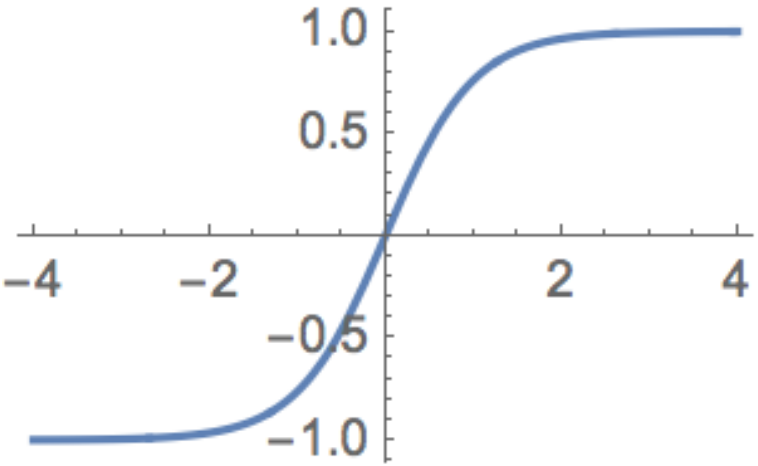
$$y = \sum_{j=0}^{2} u_{1,j}\, v_j$$

Activation Functions

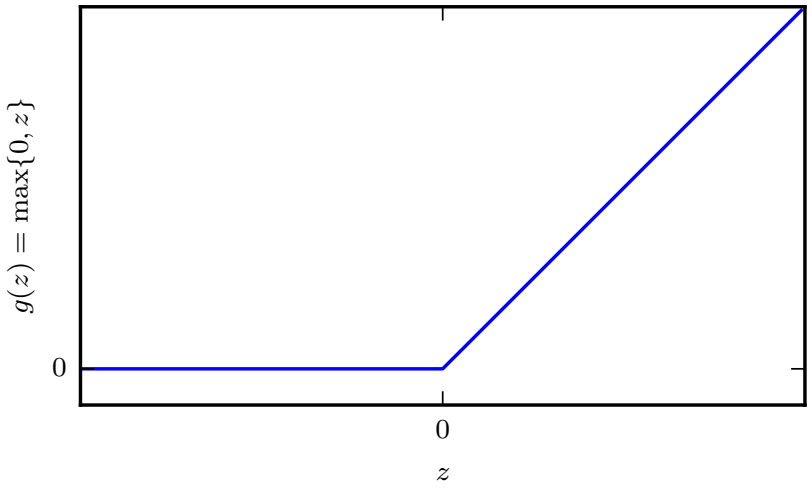
- $g(z) = \sigma(z)$



- $g(z) = \tanh(z)$ (preferred as tanh(0) = 0 and closer to identity near 0)



- $g(z) = ReLU(z)$

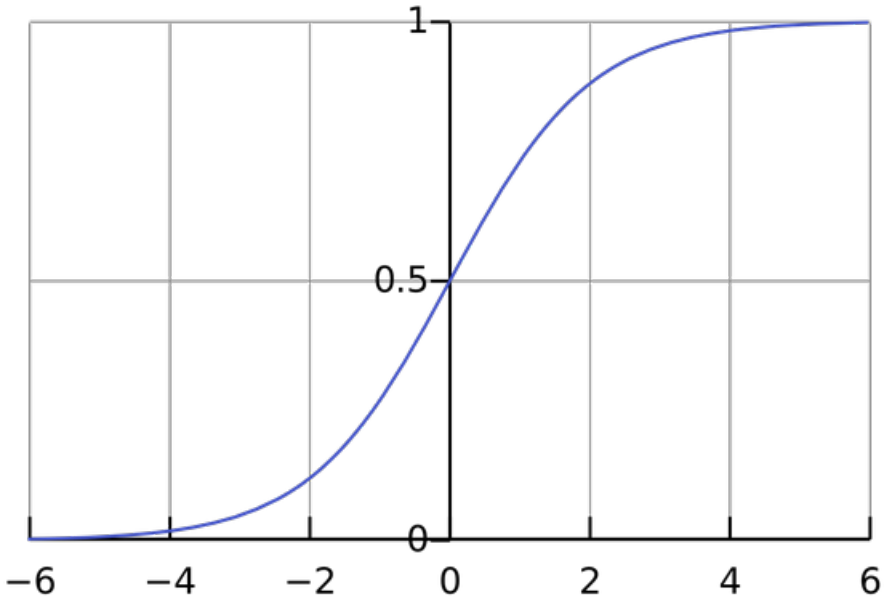


Other Options

- Softmax
- Radial Basis $g(z) = \exp\left(-\frac{1}{\sigma^2}||w - z||^2\right)$
  - "template matching"
- Softplus $\varsigma(z) = \log(1 + e^z)$
- Bounded tanh (between -1 and 1)
- Leaky ReLu
- ….

Softmax Output

- "Squashes" K-dimensional vector **z** of real values to K-dimensional vector of values in range [0.0,1.0]
  - Great for probabilities
- Represents **categorical probability distributions**: probability distribution over K possible outcomes



$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

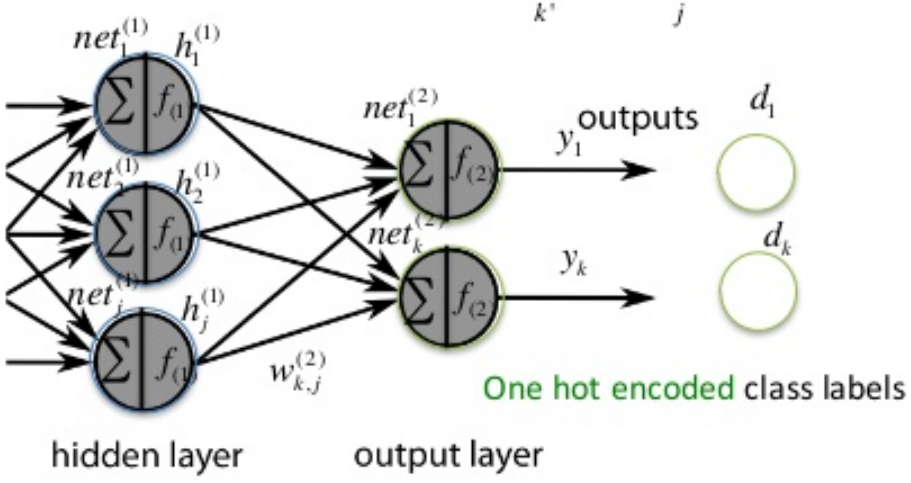
Multi-Class Classification via Softmax

Error/Loss function

- Softmax (cross-entropy loss) for multiple classes:

(Class labels follow multinomial distribution)

$$E(w) = -\sum_k (d_k \log y_k + (1 - d_k)\log(1 - y_k)) \quad \text{where} \quad y_k = \frac{\exp(\sum_j w_{k,j}^{(2)} h_j^{(1)})}{\sum_{k'} \exp(\sum_j w_{k',j}^{(2)} h_j^{(1)})}$$



"One-hot" encoding:

Suppose we have a 5-class pro blem.

A ground-truth sample labeled as class "3" would be encoded as (assumes 0-based indices):

y = [0,0,0,1,0]

How To Train Your Network

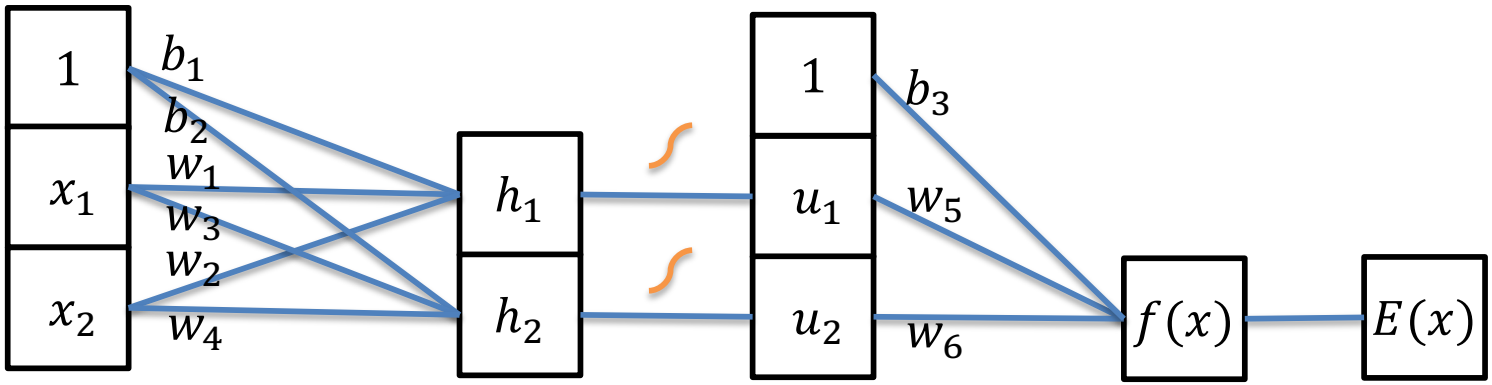
Set of N samples $\left(x^{(i)}, y^{(i)}\right)$

Define loss $\mathrm{E}(x; w) \in \mathbb{R}$ to measure error for a sample

Training: finding weights that minimize $\sum_i E(x^{(i)}; w)$ for the samples

Often uses simple gradient descent methods

Backpropagation computes the gradients of all parameters



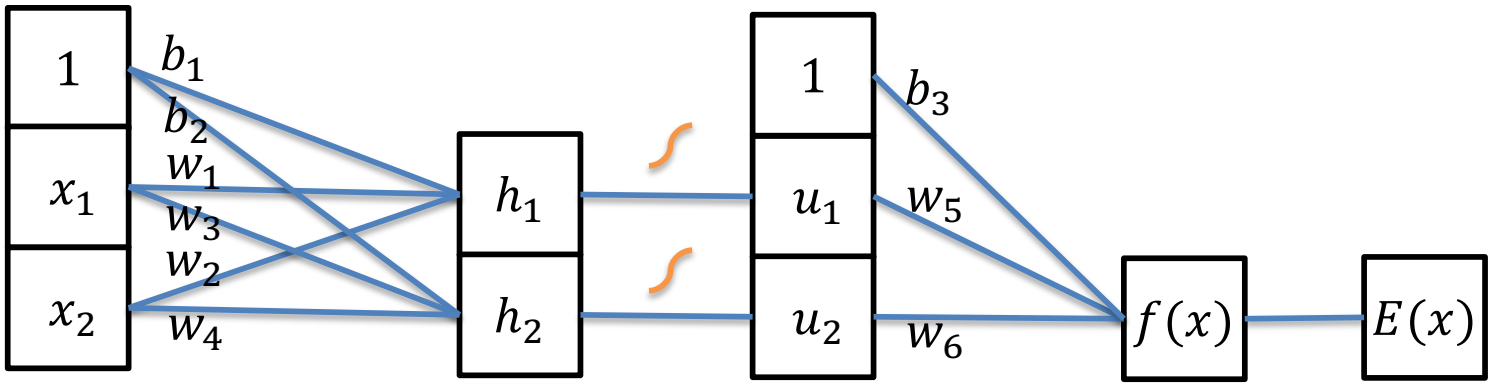
Error and Activation

Here, as an example, we choose:

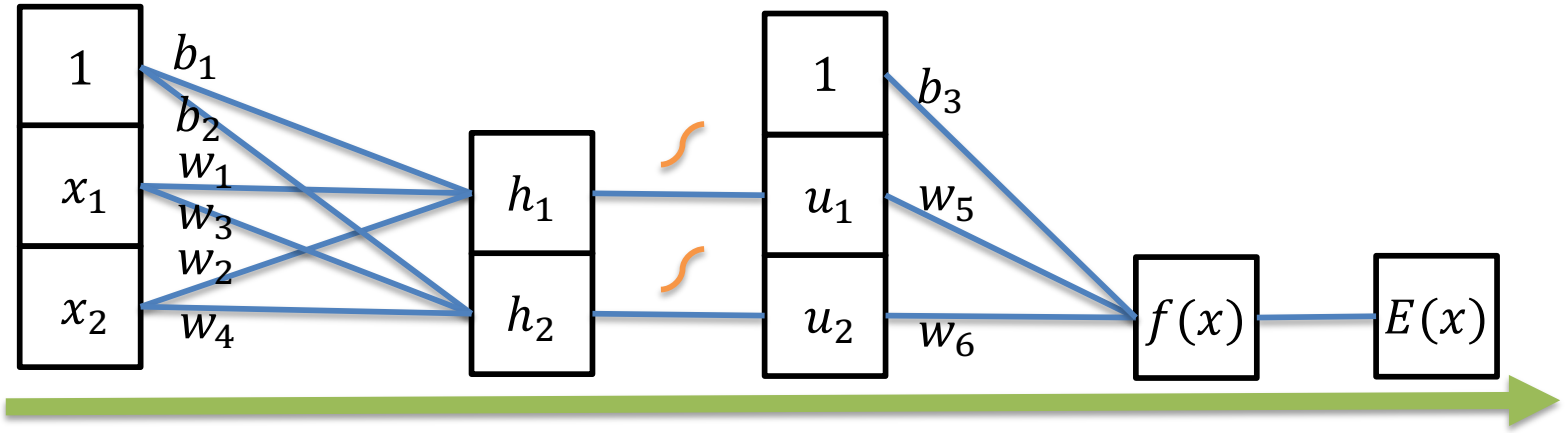
$$E\left(x^{(i)}; w\right) = \frac{1}{2}\left(f\left(x^{(i)}; w\right) - y^{(i)}\right)^2$$

$$\sigma(z) = \tanh(z)$$

BP is an efficient way to compute gradients for all parameters



Forward Pass



$$h_1 = x_1 w_1 + x_2 w_2 + b_1$$
$$h_2 = x_1 w_3 + x_2 w_4 + b_2$$

$$u_1 = \tanh h_1$$
$$u_2 = \tanh h_2$$

$$f(x) = u_1 w_5 + u_2 w_6 + b_3$$

$$E(x) = \frac{1}{2}(f(x) - y)^2$$

Backward Pass

Idea: use chain rule

$$f\left(g\left(h(x)\right)\right)$$

$$\frac{\partial}{\partial x} f\left(g\left(h(x)\right)\right) = \frac{\partial f(u)}{\partial u}\left(g\left(h(x)\right)\right)\frac{\partial g(v)}{\partial v}\left(h(x)\right)\frac{\partial h(x)}{\partial x}(x)$$

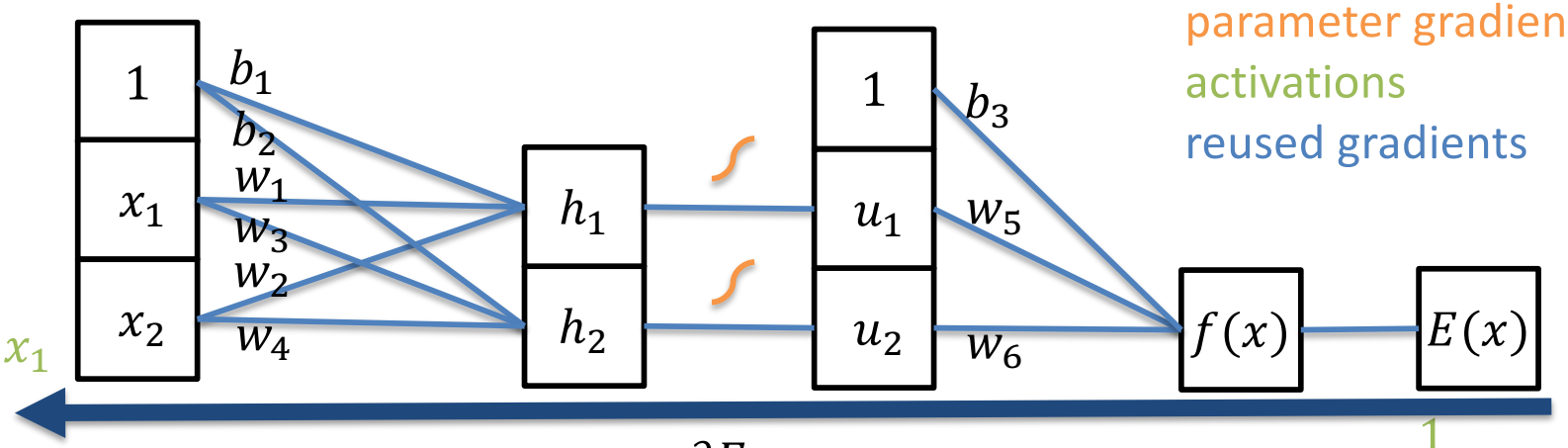
simpler (Leibnitz's) notation:

$$t = f(u), \qquad u = g(v), \qquad v = h(x)$$

$$\frac{\partial t}{\partial x} = \frac{\partial t}{\partial u}\frac{\partial u}{\partial v}\frac{\partial v}{\partial x}$$

Backward Pass

parameter gradients
activations
reused gradients



$$\frac{\partial E}{\partial b_1} = \delta_4$$

$$\frac{\partial E}{\partial w_1} = \delta_4 x_1$$

$$\frac{\partial E}{\partial w_2} = \delta_4 x_2$$

$$h_1 = x_1 w_1 + x_2 w_2 + b_1$$

$$h_2 = x_1 w_3 + x_2 w_4 + b_2$$

$$\frac{\partial E}{\partial u_1} = \delta_1 w_5 = \delta_2$$

$$\frac{\partial E}{\partial u_2} = \delta_1 w_6 = \delta_3$$

$$E(x) = \frac{1}{2}(f(x) - y)^2$$

$$\frac{\partial E}{\partial f(x)} = f(x) - y = \delta_1$$

$$\frac{\partial E}{\partial b_2} = \delta_5$$

$$\frac{\partial E}{\partial w_3} = \delta_5 x_1$$

$$\frac{\partial E}{\partial w_4} = \delta_5 x_2$$

$$u_1 = \tanh h_1$$

$$u_2 = \tanh h_2$$

$$f(x) = u_1 w_5 + u_2 w_6 + b_3$$

$$\frac{\partial E}{\partial h_1} = \frac{\partial E}{\partial u_1}\frac{\partial u_1}{\partial h_1} = \delta_2(1 - \tanh^2 h_1) = \delta_4$$

$$\frac{\partial E}{\partial h_2} = \frac{\partial E}{\partial u_2}\frac{\partial u_2}{\partial h_2} = \delta_3(1 - \tanh^2 h_2) = \delta_5$$

$$\frac{\partial E}{\partial b_3} = \frac{\partial E}{\partial f}\frac{\partial f}{\partial b_3} = \delta_1 \cdot 1$$

$$\frac{\partial E}{\partial w_5} = \frac{\partial E}{\partial f}\frac{\partial f}{\partial w_5} = \delta_1 u_1$$

$$\frac{\partial E}{\partial w_6} = \frac{\partial E}{\partial f}\frac{\partial f}{\partial w_6} = \delta_1 u_2$$

Gradient Descent

Initialize all weights (randomly, close to solution, …)

How to use the derivatives?

- Gradient descent to minimize error function

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

Update the weights in every iteration of training with a small gradient step

Learning rate λ adjusts the stepsize

Stochastic Gradient Descent

Error was defined over the whole training set: $\sum_i E(x^{(i)})$

Need to compute and sum the derivatives of all samples before gradient step

Slow but accurate updates

Stochastic Gradient Descent (SGD): approximate derivative from small, random subset ([mini]batch) of training set

Noisy but faster

Usually: make sure to see every sample the same amount of times (epochs)

Momentum

Narrow ravines in error function make gradient descent methods oscillate

Slow convergence

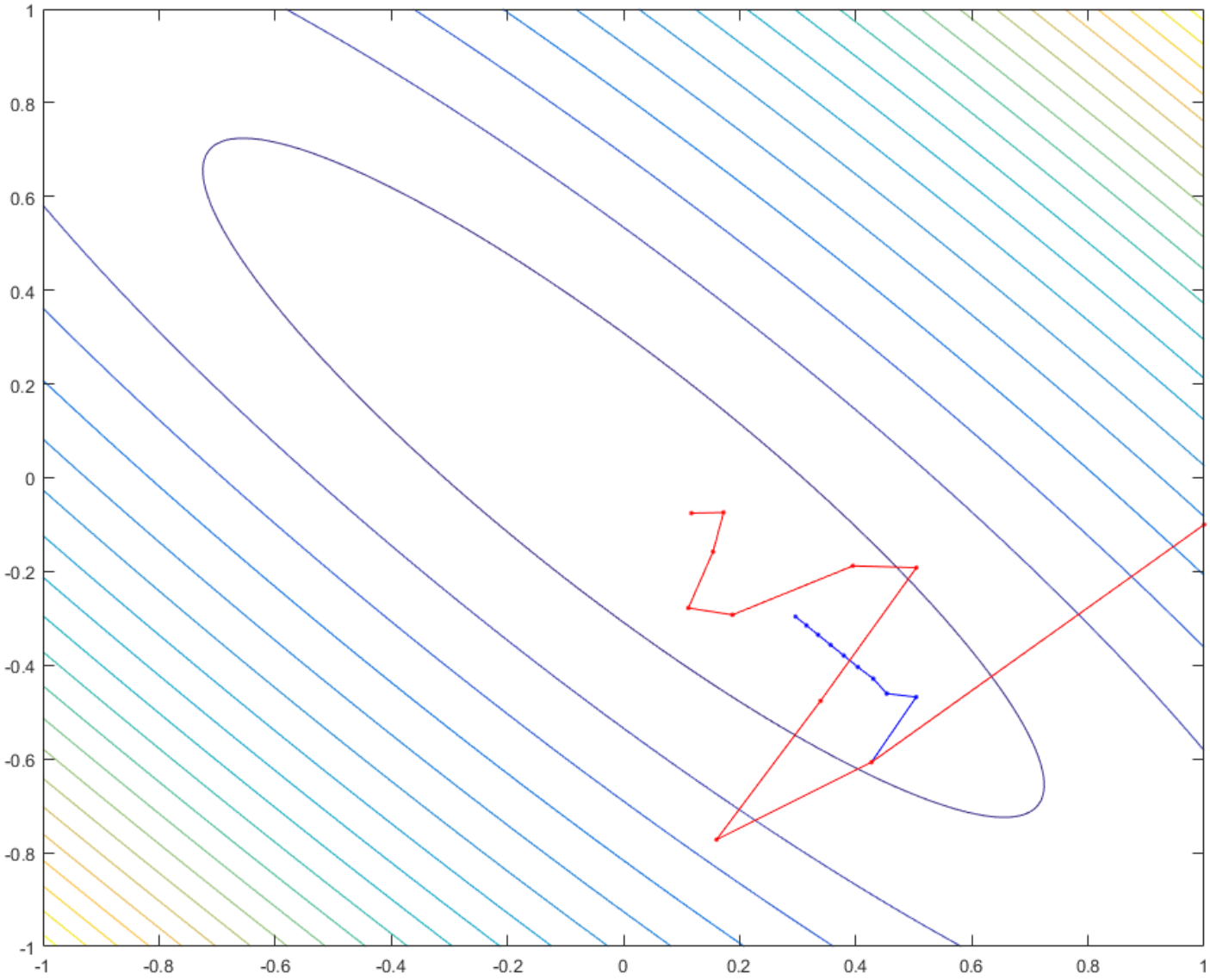
Momentum: velocity v that accumulates gradients

$$v^{(t+1)} = \alpha v^{(t)} + \lambda \frac{\partial E}{\partial w}$$
$$w^{(t+1)} = w^{(t)} - v^{(t+1)}$$

$0 \leq \alpha < 1$: how much velocity is kept each step
(usually between 0.5 and 0.9)

Also accelerates learning on plateaus

Momentum



Vanishing Gradients

Activations σ have a problem

$$\left|\frac{\partial \sigma(x)}{\partial x}\right| \leq 1$$

Chain rule forces multiplications

Every layer with an activation function is almost guaranteed to scale down the incoming gradient δ

First layers learn very slowly

Makes it difficult to train deep architectures

Current solutions: pre-training, other σ, (batch) normalization, skip connections

Fitting, Models, Generalization

- When will MSE_{train} approach MSE_{test}?
  - When E and R are statistically identical: i.i.d from same distribution p(y | x)

- In general, we never have this, so we seek two things
  1. Make MSE_{train} small
  2. Make | MSE_{train} – MSE_{test} | small

- If we don't accomplish 1, we underfit
- If we don't accomplish 2, we overfit

Linear Regression

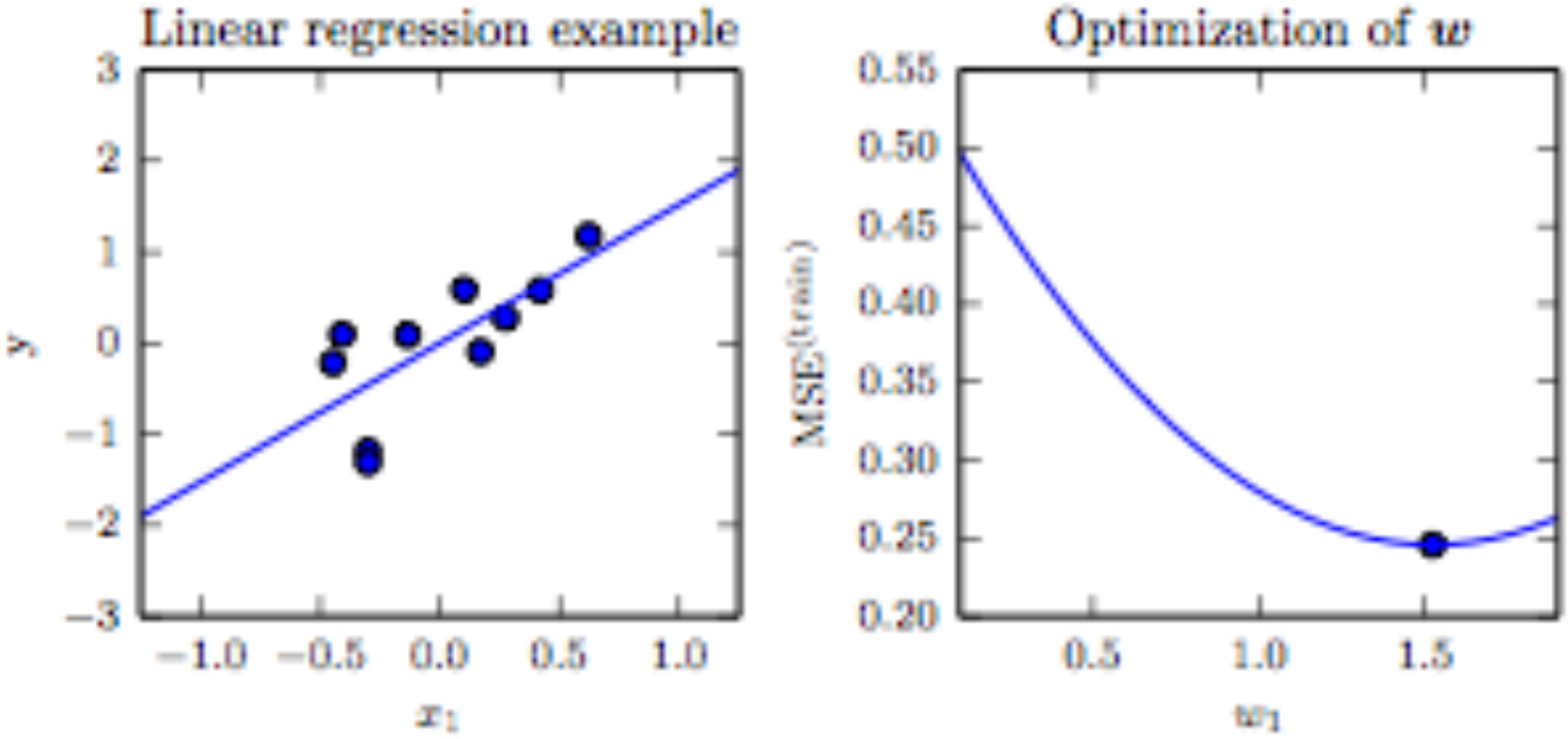


Figure 5.1

What the Right Mix of Train and Test?

- What if I have 10 points
  - Suppose I use 1 for training and 9 for testing?
  - Suppose I use 2 for training and 8 for testing?
  - ….
  - Suppose I use 9 for training and 1 for testing?

- Do I need to consider only linear models?
  - $y = \sum_{i=0}^{n} a_i x^i$
  - How to choose n?

Underfitting and Overfitting in Polynomial Estimation

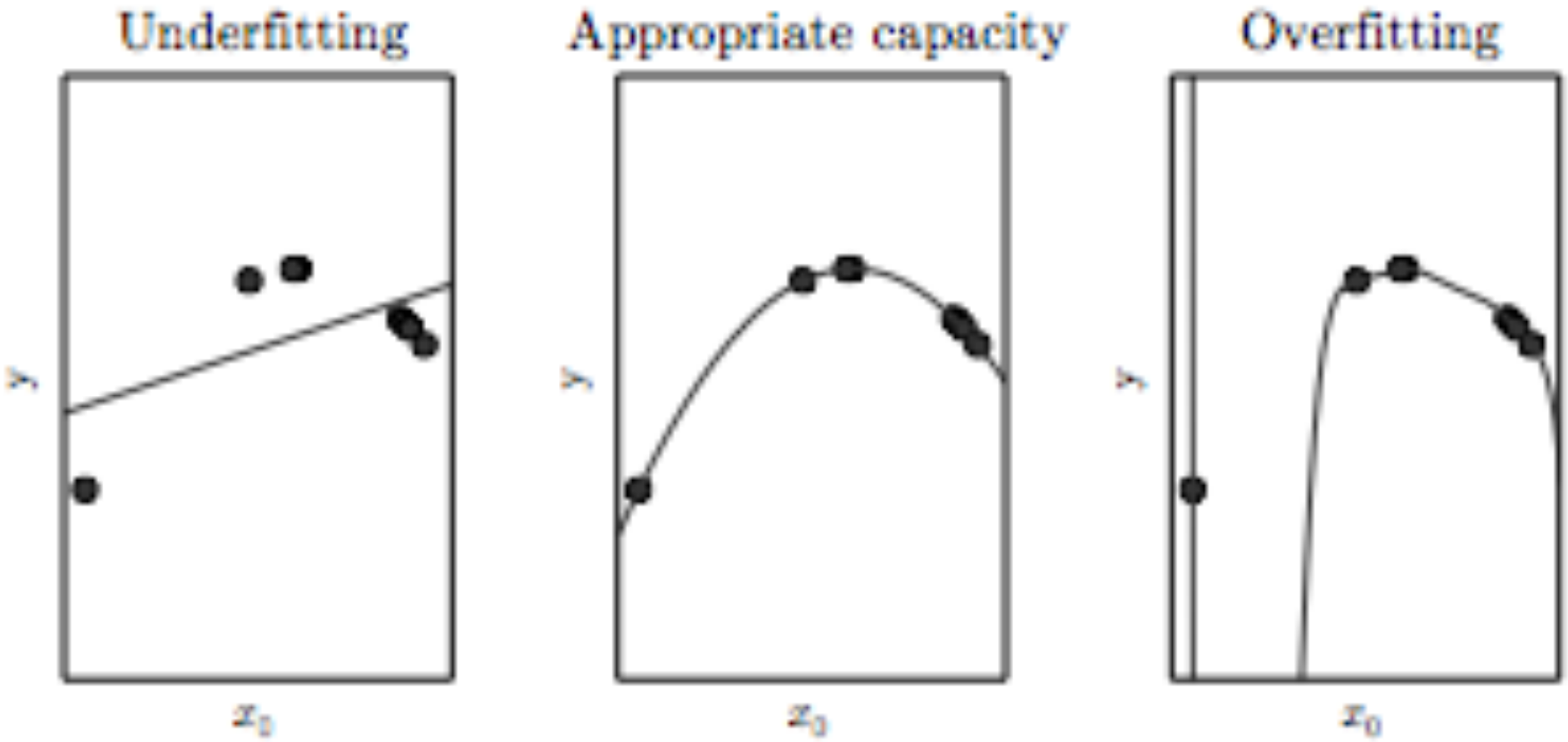
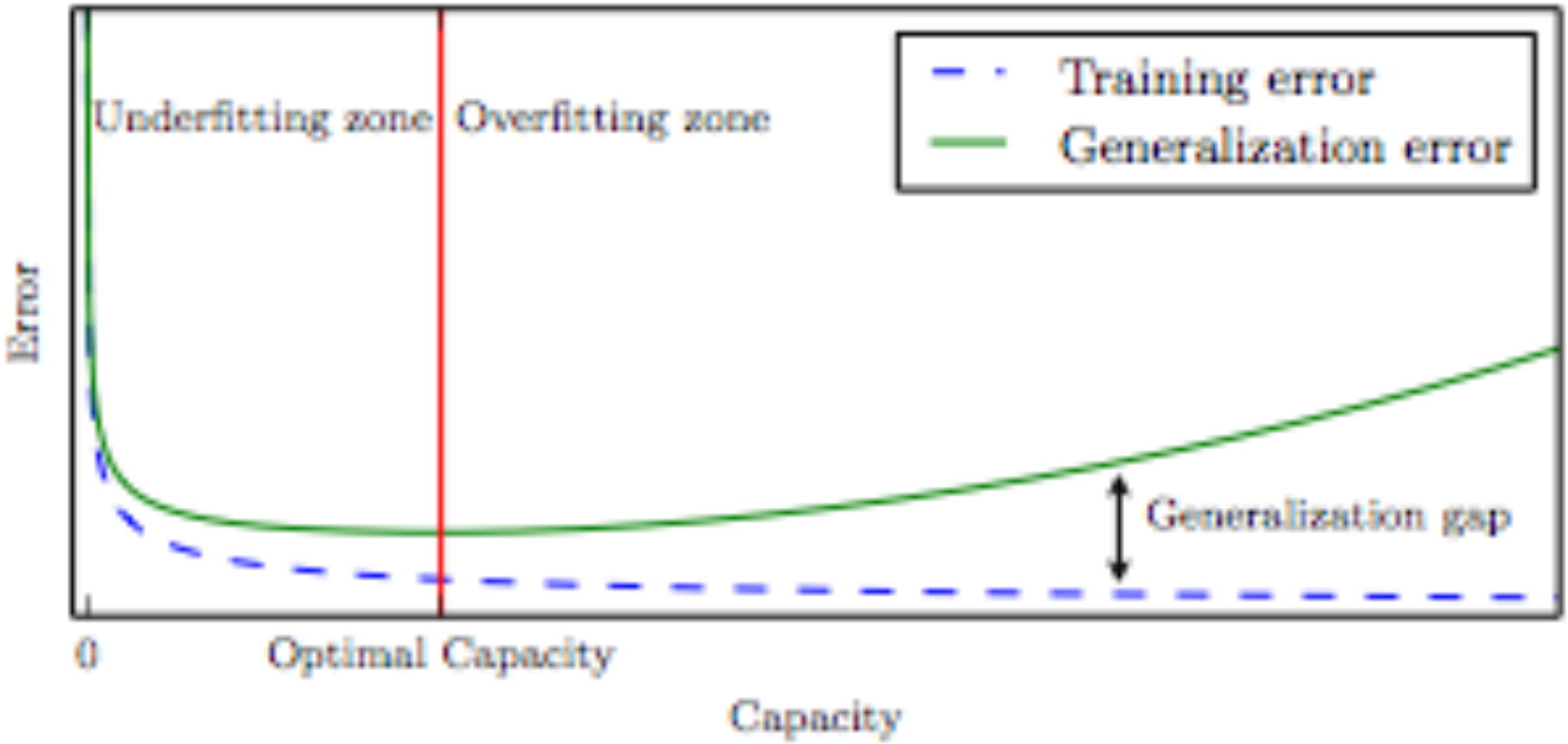


Figure 5.2

Generalization and Capacity



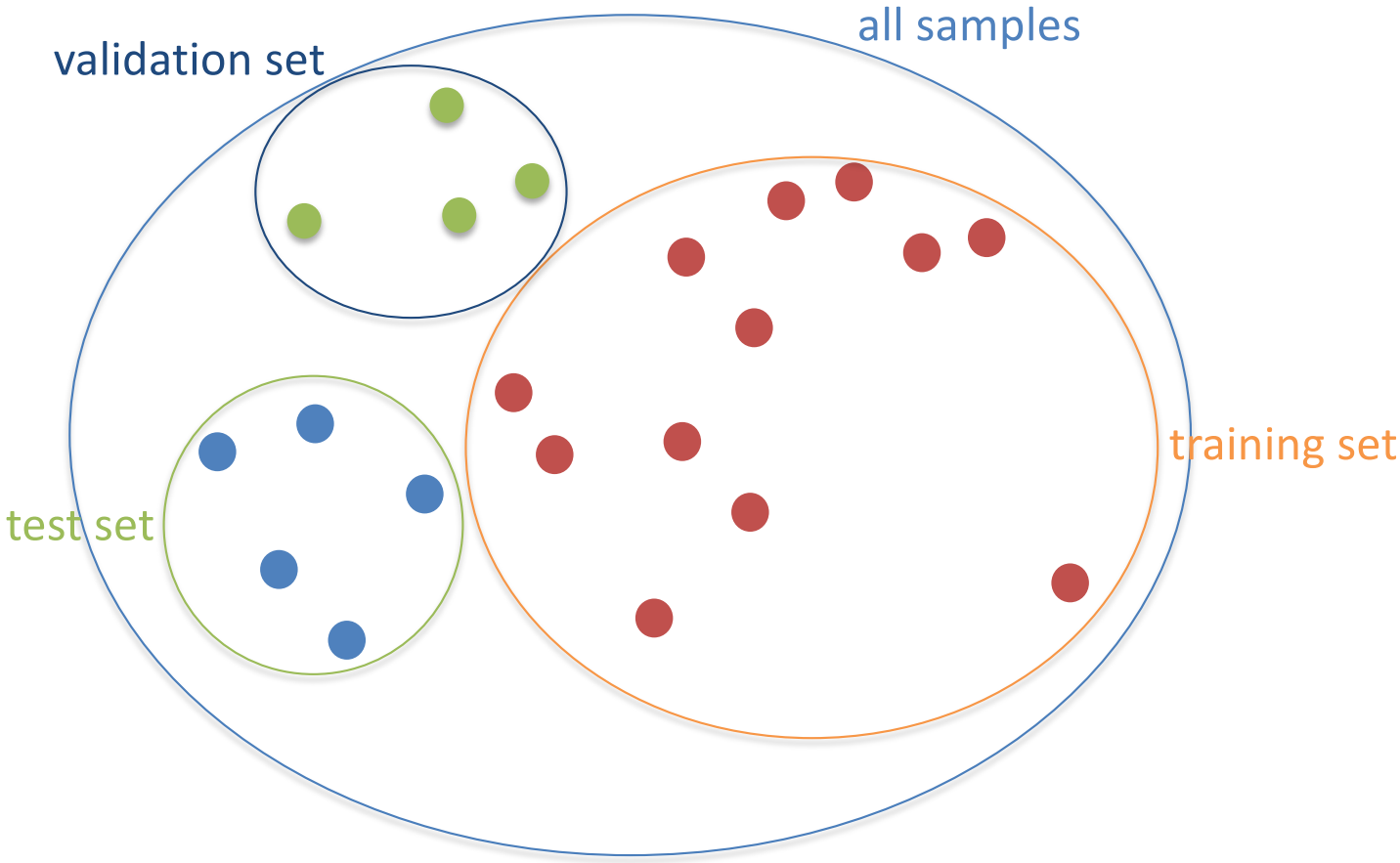
Welcome to HyperParameters

- We now have a “problem in a problem”
  - Choose parameters
  - Optimize
  - Check if model is good → i.e. it generalizes
  - Repeat

- Now we have to create a *validation set* within the training set

- Common to use cross-validation (repeated sampling, training validation) within training set.

Training – Validation - Testing



Ensuring Good Generalization

- A common issue with models with very large numbers of parameters
  - Basically memorizing the training data and not generalizing well
- One trick is to combine predictions from different models trained on the same data
  - With Deep CNNs, this would take way too long to train (and too much memory)
- However, there's a "trick" we can use: **Regularization**

Regularization: Expressing Preference

$$L(W) = \underbrace{\frac{1}{N} \sum_i L_i \left(f(x_i, W), y_i \right)}$$

Data term:
Predictions must match annotations

Regularization: Expressing Preference

$$L(W) = \underbrace{\frac{1}{N}\sum_i L_i\left(f(x_i, W), y_i\right)}_{\text{Data term}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data term:
Predictions must match annotations

Regularization
λ strength (hyperparameter)

Regularization: Expressing Preference

$$L(W) = \underbrace{\frac{1}{N}\sum_i L_i\left(f(x_i, W), y_i\right)}_{\text{Data term}} + \lambda R(W)$$

Data term:
Predictions must
match annotations

→ Loss function L(W) must accurately reflect our desired behavior of f(x,W)!

- Data loss
  - This is straight forward
- Regularization
  - Introduce preferences on weights; e.g. sparse or of small magnitude
  - Counteract overfitting by enforcing simpler models
  - Aid optimization (see later) by shaping the loss function (adding curvature)

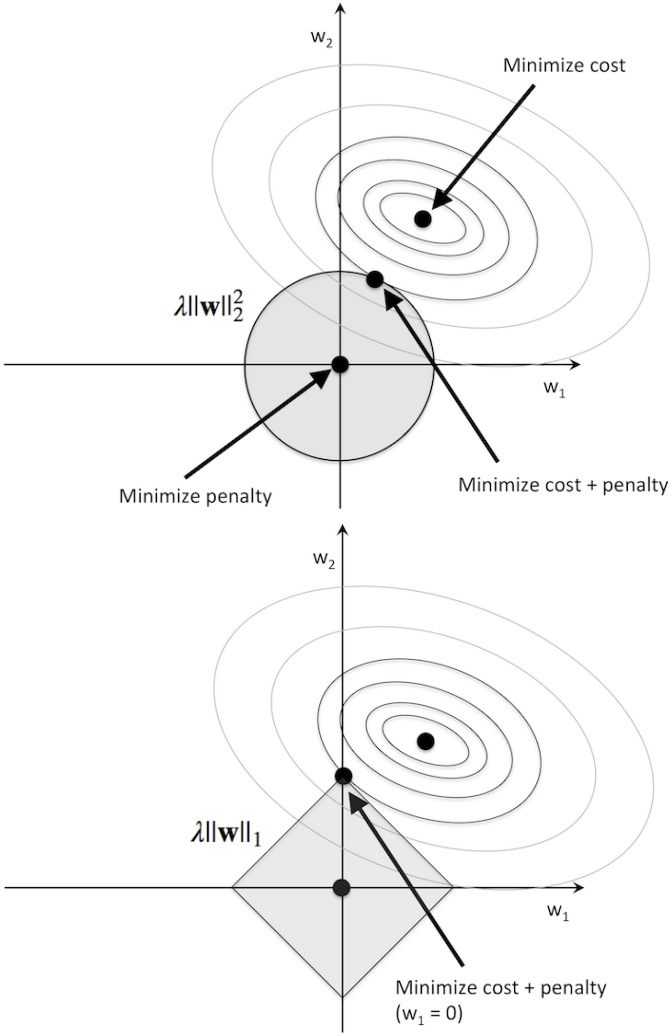
Regularization: Examples

$$L(W) = \underbrace{\frac{1}{N}\sum_i L_i\left(f(x_i, W), y_i\right)}_{} + \lambda R(W)$$

Data term:
Predictions must
match annotations

Simple regularizers

- L2 (magnitude): $R(W) = \sum_{k,l} W_{k,l}^2$
- L1 (sparsity): $R(W) = \sum_{k,l} |W_{k,l}|$
- Versions, e.g. Elastic net, Huber,…



Images from rasbt.github.io.

Regularization: Examples

$$L(W) = \underbrace{\frac{1}{N}\sum_i L_i\left(f(x_i, W), y_i\right)}_{\text{Data term}} + \lambda R(W)$$

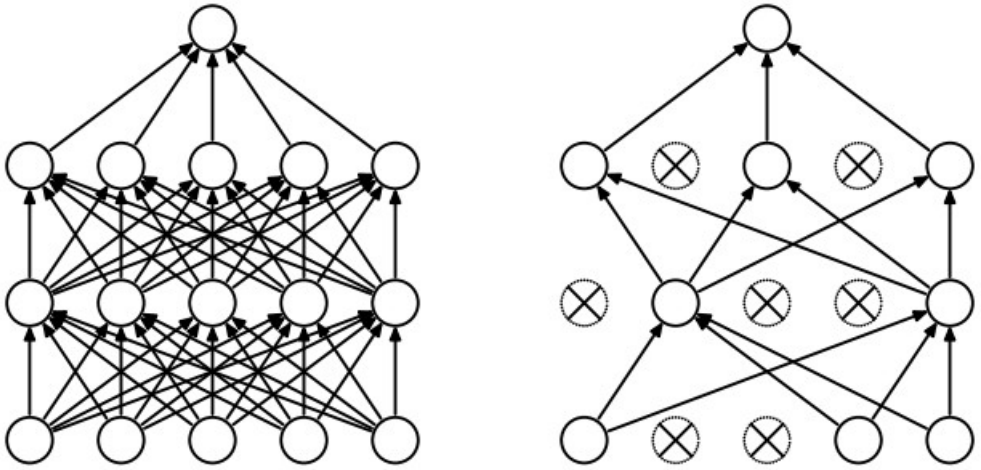
Data term:
Predictions must
match annotations

More complex regularizers

- Dropout [1]
- Batch normalization [2]
- Stochastic depth [3], … and many others

[1] Srivastava, N., et al (2014). Dropout: a simple way to prevent neural networks from overfitting. JMLR
[2] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv
[3] Huang, G., et al. (2016) Deep networks with stochastic depth. ECCV

Dropout



(a) Standard Neural Net

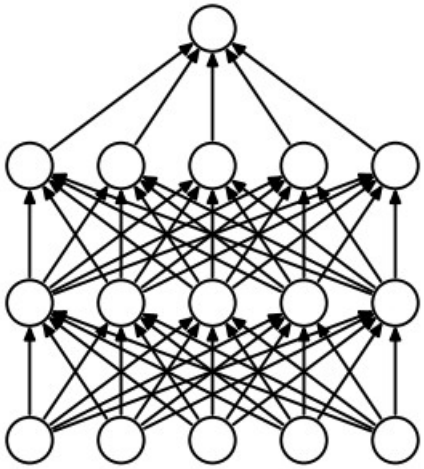
(b) After applying dropout.

Basic idea: during training, with some probability (say 0.5), set the output of each node of a fully-connected layer to zero.

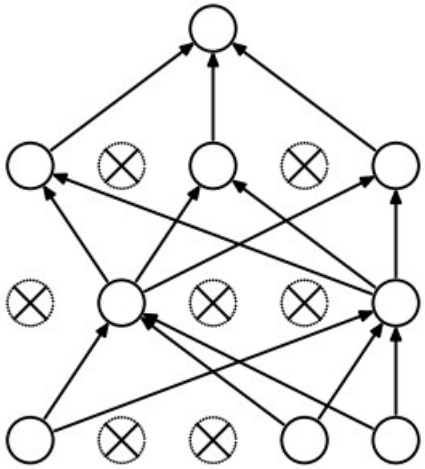
In this way, the neurons that are "dropped out" don't contribute to that solution.

Each time a sample is presented, the network samples a different architecture to solve the problem, though all weights are shared

Dropout



(a) Standard Neural Net



(b) After applying dropout.

Reduces “co-adaptations of neurons”

Now, a neuron cannot rely on the presence of some other particular neuron(s)

Forced to learn more robust features that are useful with many different subsets of the other neurons

At test time, we use all neurons but multiply outputs by 0.5 which reasonably approximates the geometric mean

Scanning for Patterns

Convolutional Neural Networks

Convolutional Networks: Some Motivating Problems

- Segmentation
- Recognition
- Pose estimation
- Scene captioning
- Video segmentation

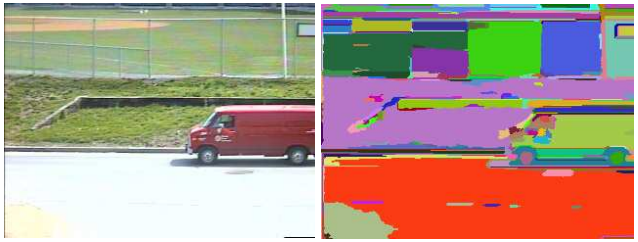


Figure 2: A street scene (320×240 color image), and the segmentation results produced by our algorithm ($\sigma = 0.8$, $k = 300$).

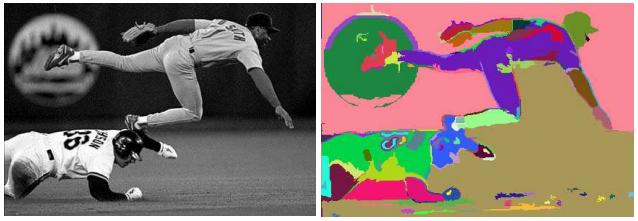


Figure 3: A baseball scene (432×294 grey image), and the segmentation results produced by our algorithm ($\sigma = 0.8$, $k = 300$).

Figure 4: An indoor scene (image 320×240, color), and the segmentation results produced by our algorithm ($\sigma = 0.8$, $k = 300$).

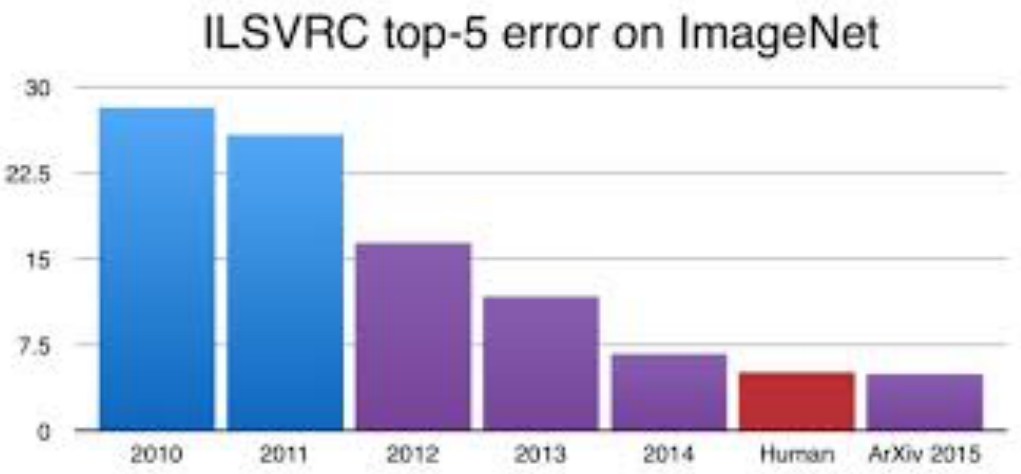
From Felzenszwalb

Convolutional Networks: Some Motivating Problems

- Segmentation
- Recognition
- Pose estimation
- Scene captioning
- Video segmentation

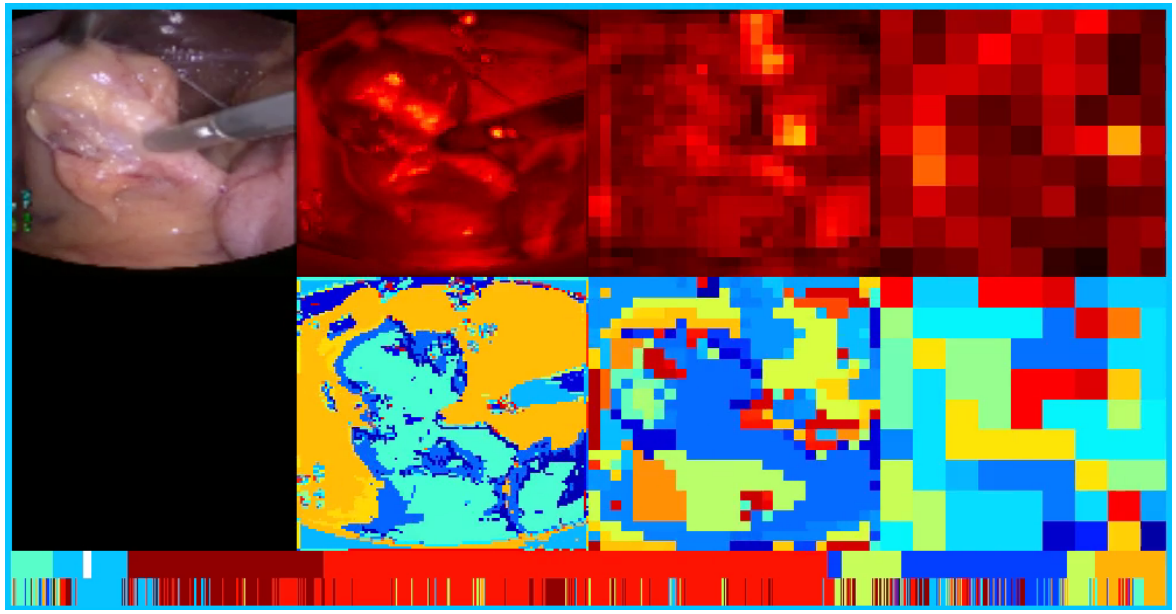
[Deng et al. CVPR 2009]

14M Images, 20k classes



Convolutional Networks: Some Motivating Problems

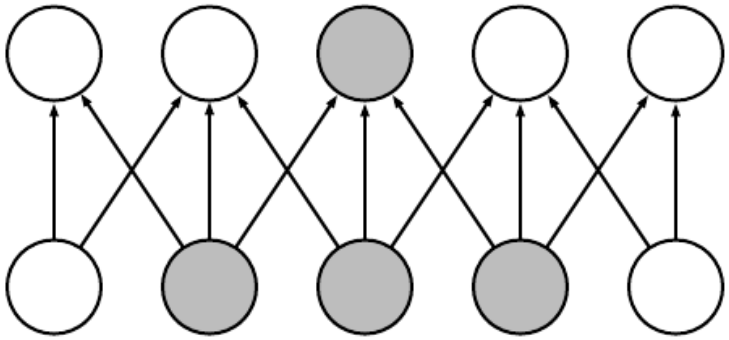
- Segmentation
- Recognition
- Pose estimation
- Scene captioning
- Video segmentation



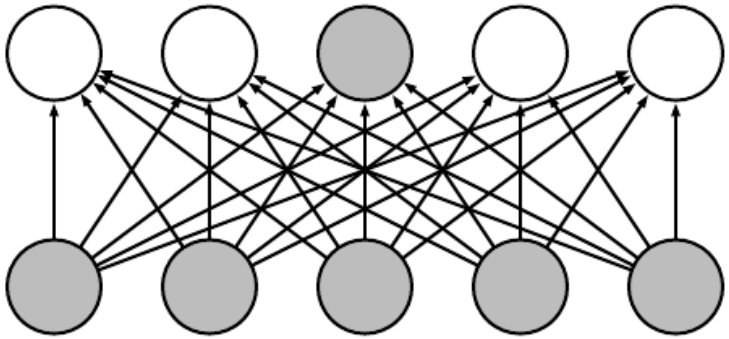
DiPietro, Robert, et al. "Recognizing surgical activities with recurrent neural networks." *MICCAI*, 2016.
Lea, Colin, et al. "Temporal Convolutional Networks for Action Segmentation and Detection." *CVPR*. 2017.

The Common Element: Convolution

Convolution = sparse, shared parameters



1kx1k bw image
5x5 convolution
20 channels
500 parameters (!)



1kx1k bw image
Fully connected layers
10^12 parameters!

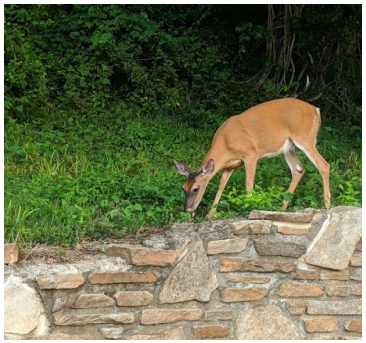
The Image Classification Problem: Revisited

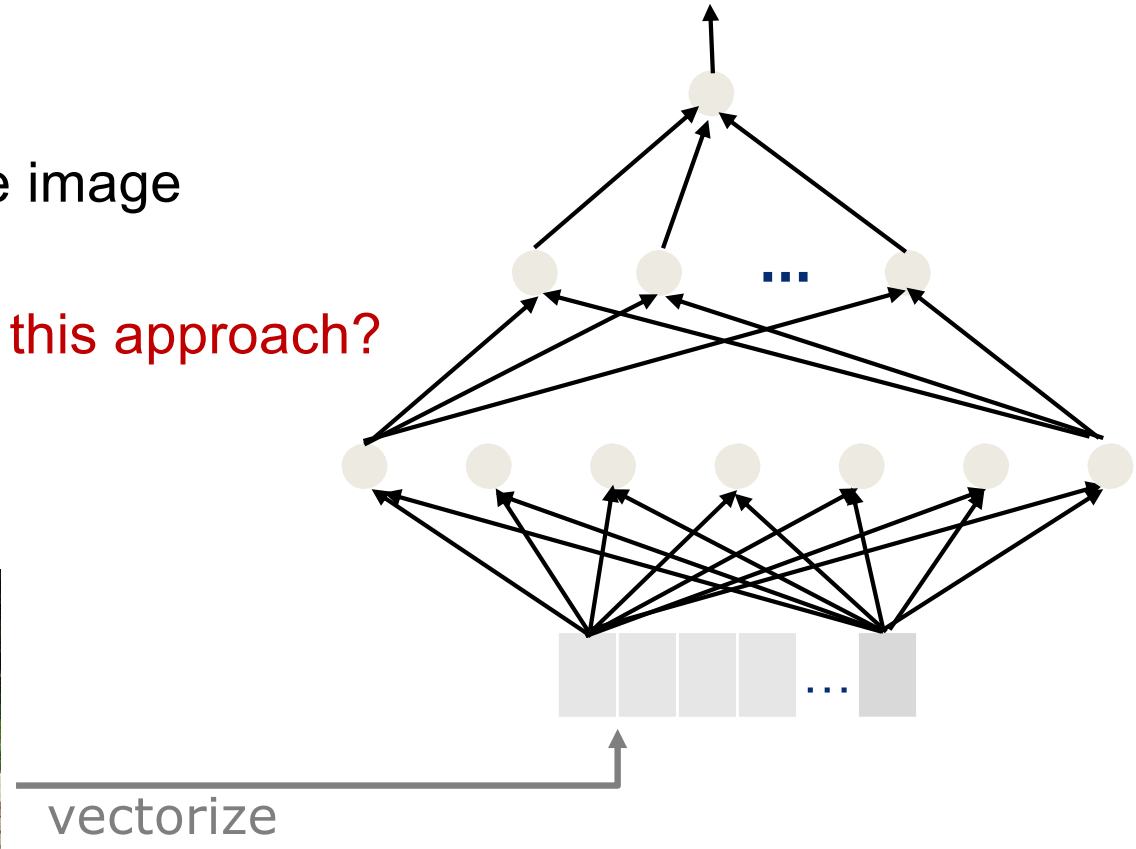
Is there a deer in this image?

Trivial solution:

→ Train a MLP for the entire image

Q: What is the problem with this approach?





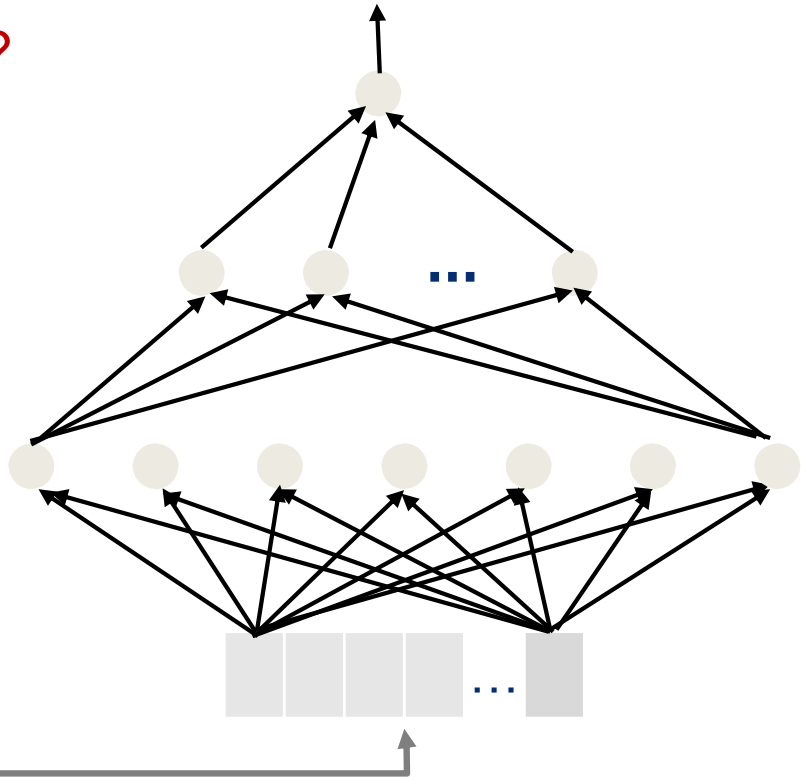
Adapted from Unberath

The Image Classification Problem: Revisited

Q: What is the problem with this approach?

An MLP trained on the right image will not find a deer in the left one (unless trained on both).

→ Large amount of training data!



Adapted from Unberath

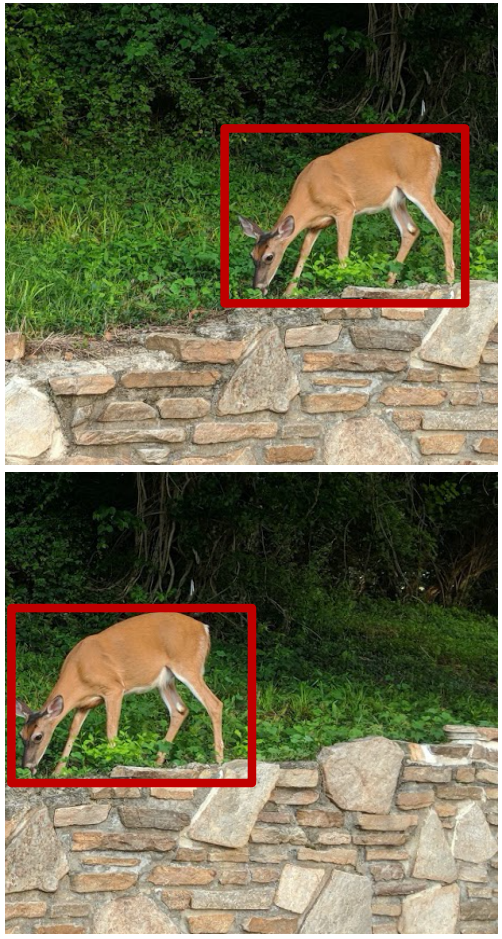
The Image Classification Problem: Revisited

What we need / want:

→ Simple solution that works on both!

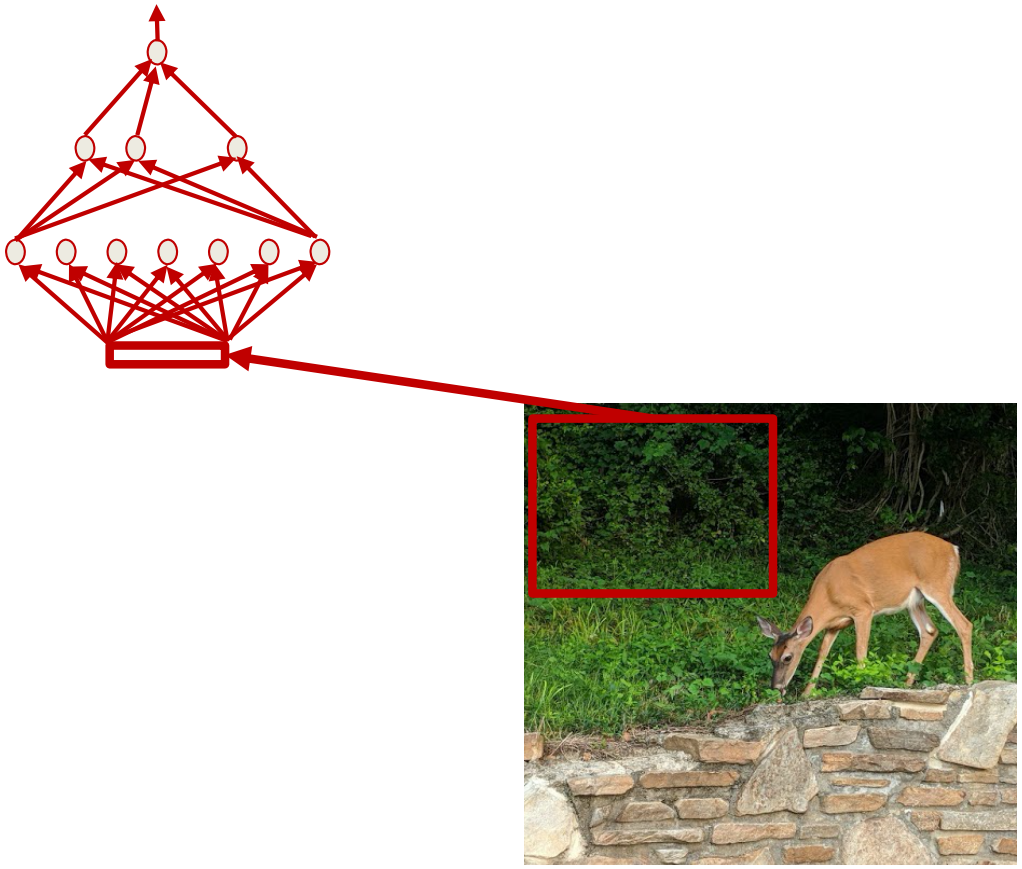
Conventional MLPs are sensitive to location, but often the **location** of a pattern **is not important**!

→ Shift invariance!



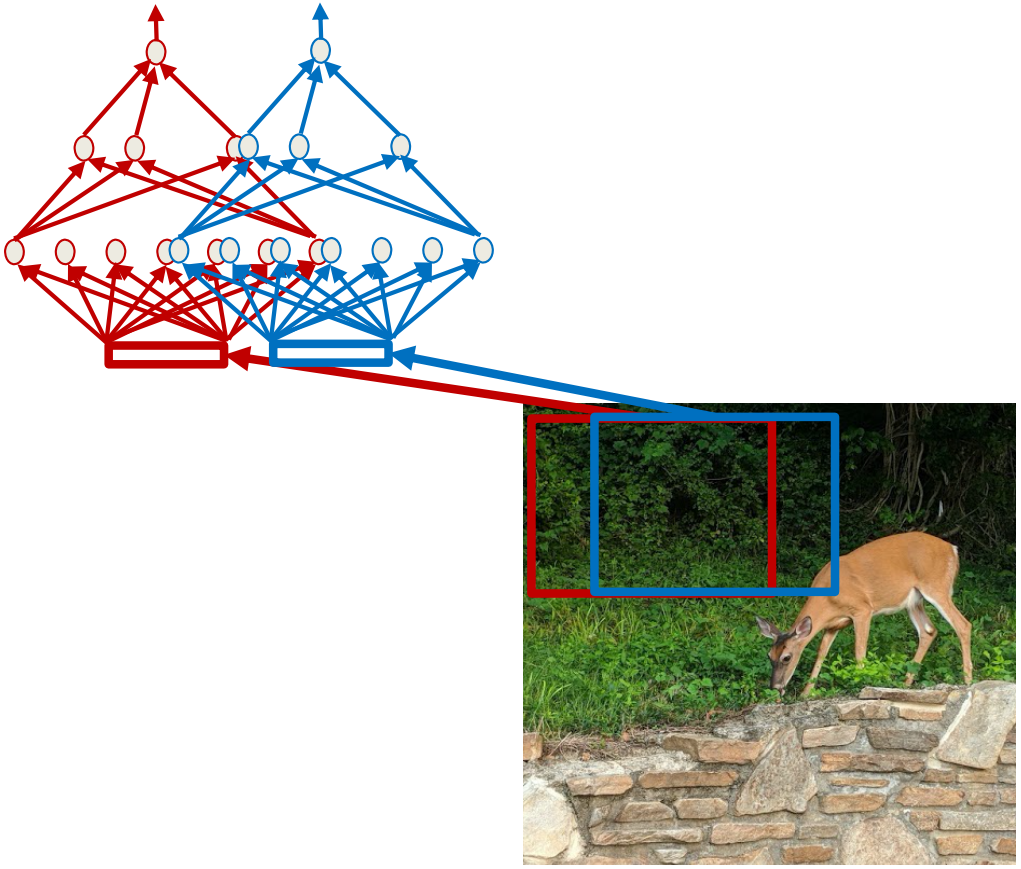
Adapted from Unberath

Shift Invariance: Scanning for Patterns



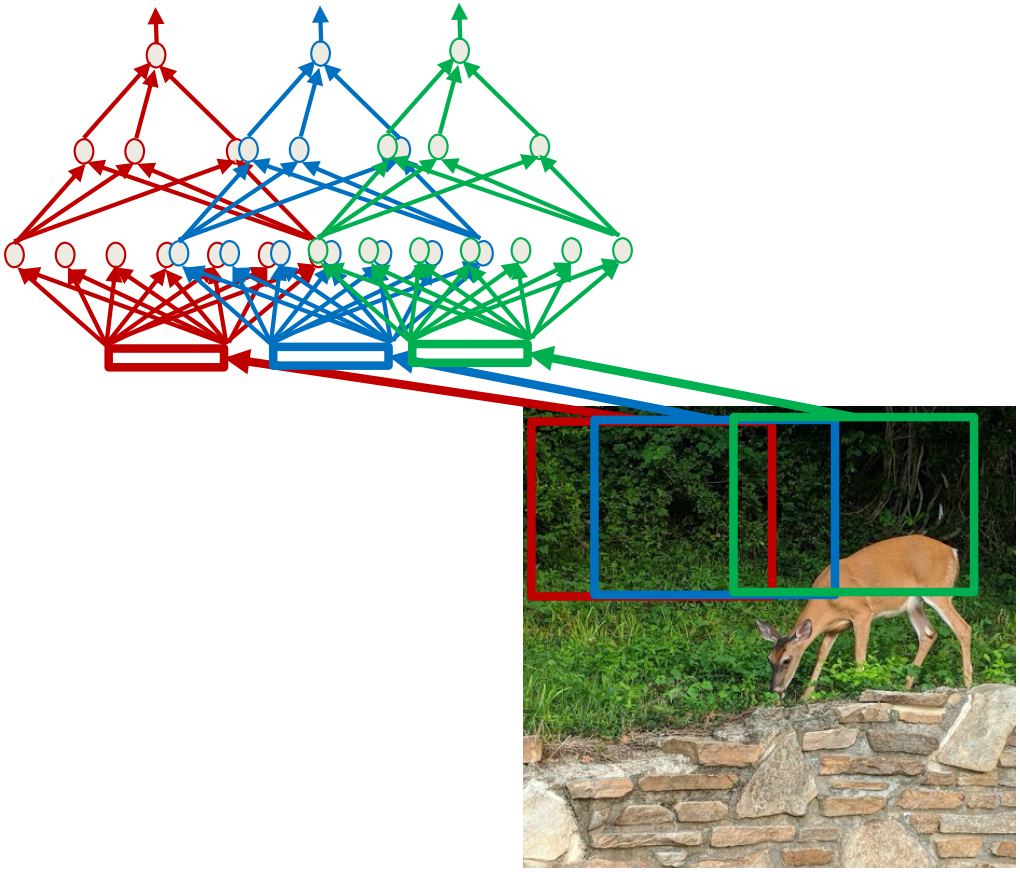
Adapted from Unberath

Shift Invariance: Scanning for Patterns



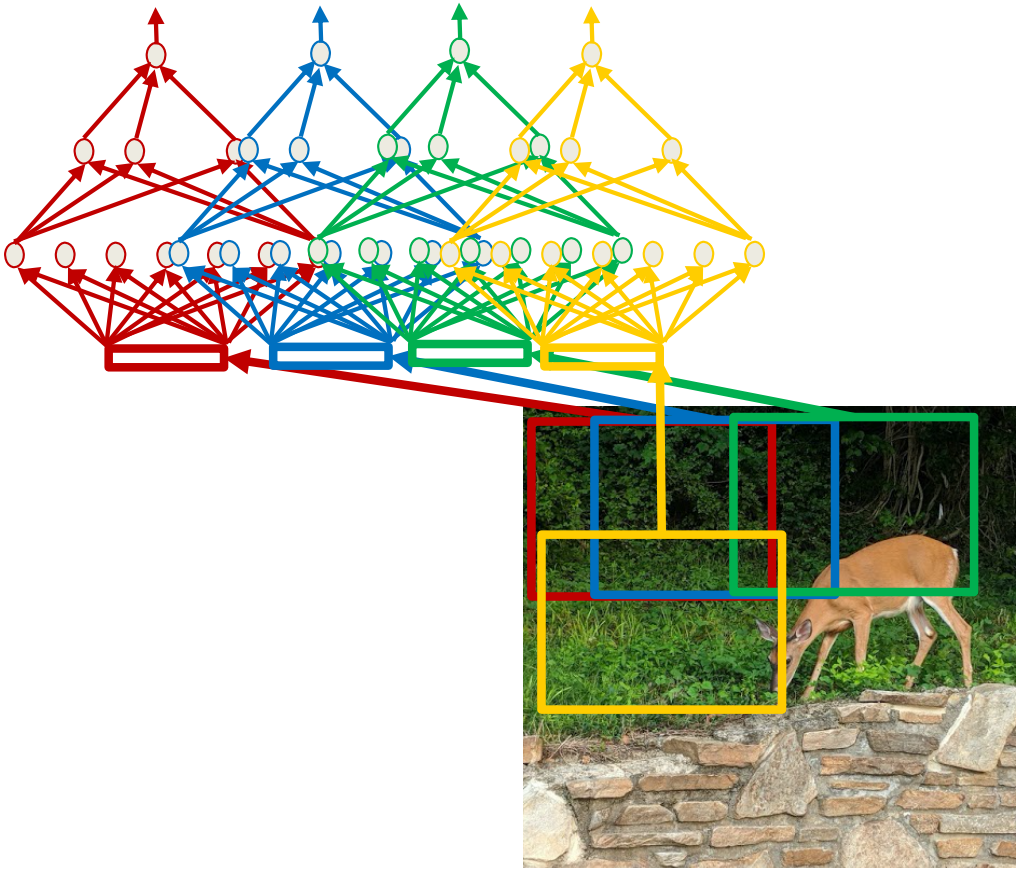
Adapted from Unberath

Shift Invariance: Scanning for Patterns



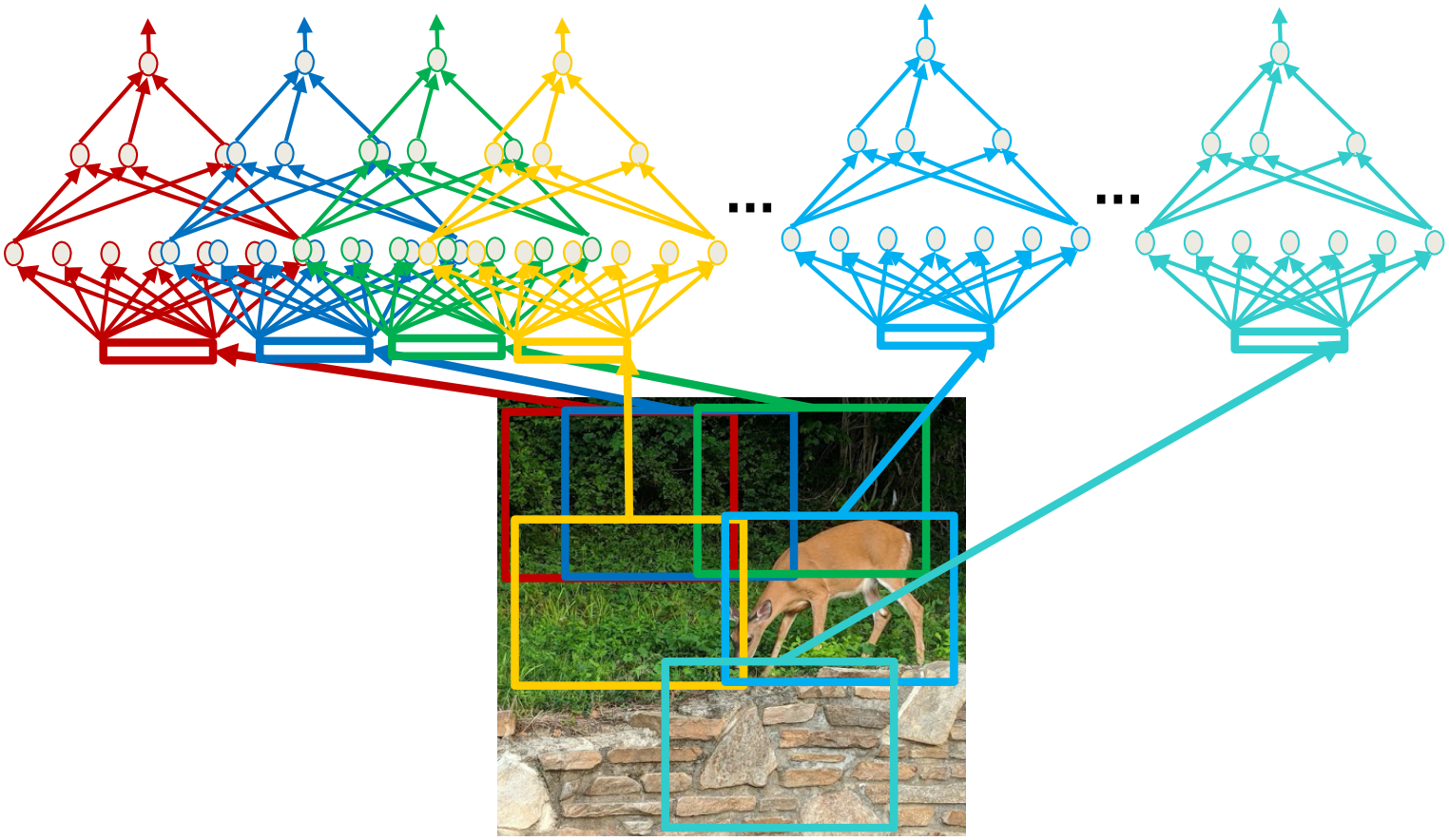
Adapted from Unberath

Shift Invariance: Scanning for Patterns



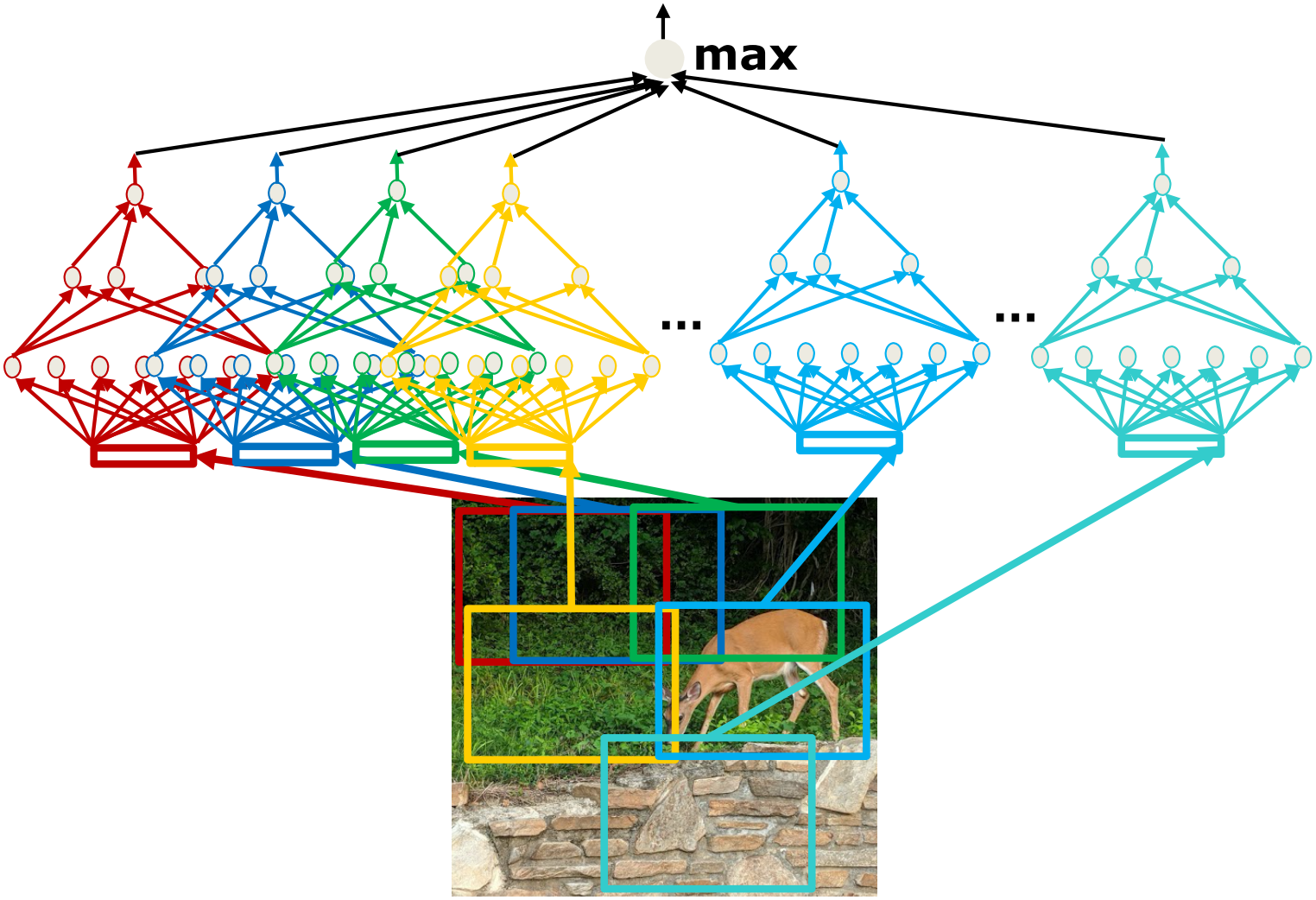
Adapted from Unberath

Shift Invariance: Scanning for Patterns



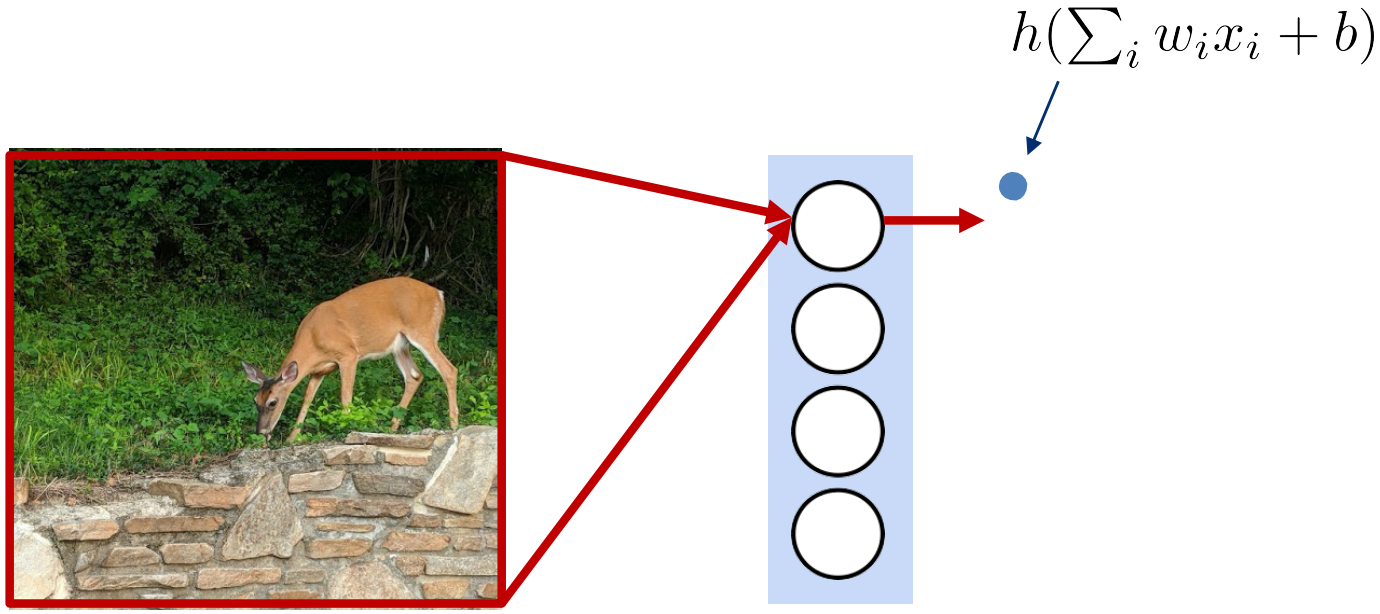
Adapted from Unberath

Shift Invariance: Scanning for Patterns



Adapted from Unberath

Scanning: A Closer Look

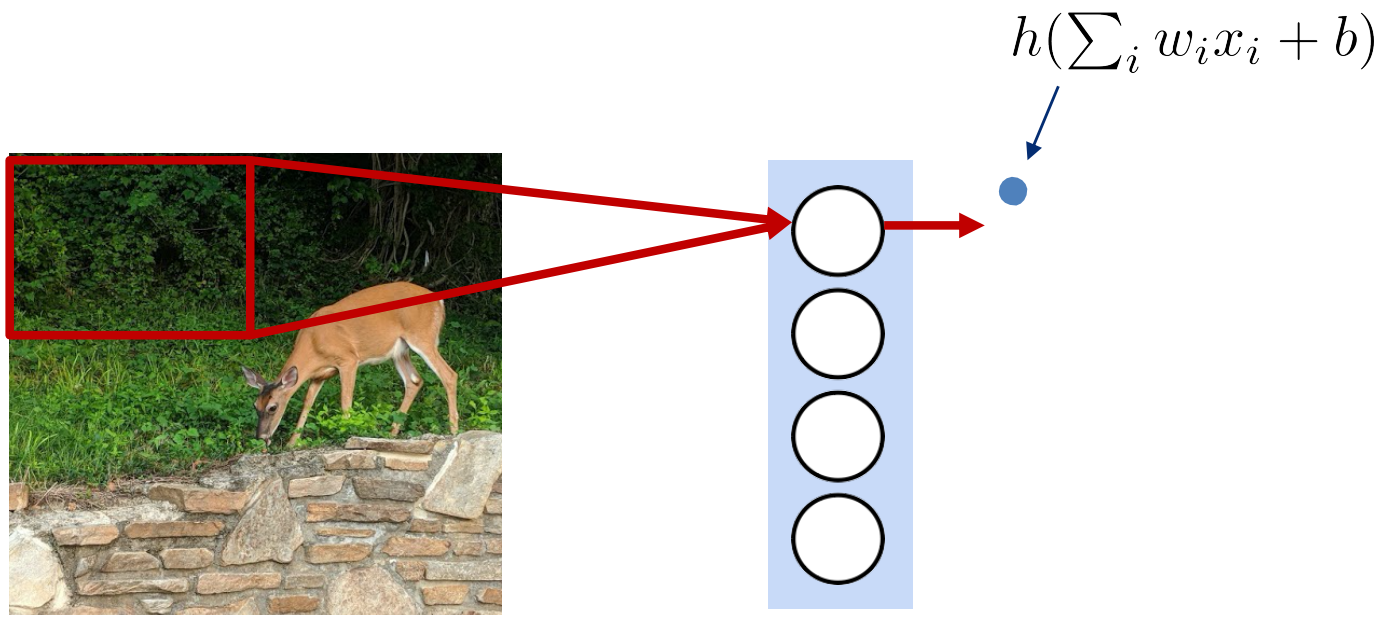


Previously

→ Evaluate perceptron on complete image

Adapted from Unberath

Scanning: A Closer Look



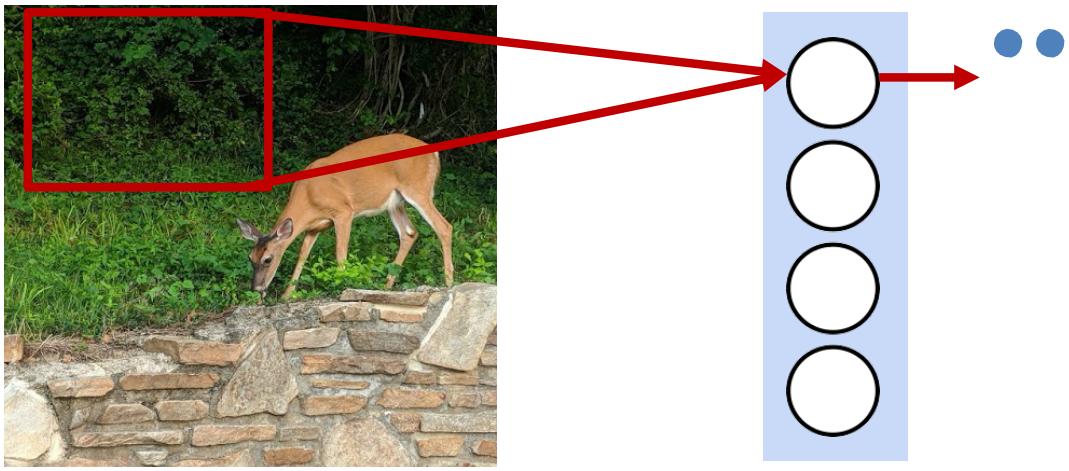
Now

→ Perceptron evaluates this region of the image

→ We can arrange these outputs in form of the original picture

Adapted from Unberath

Scanning: A Closer Look



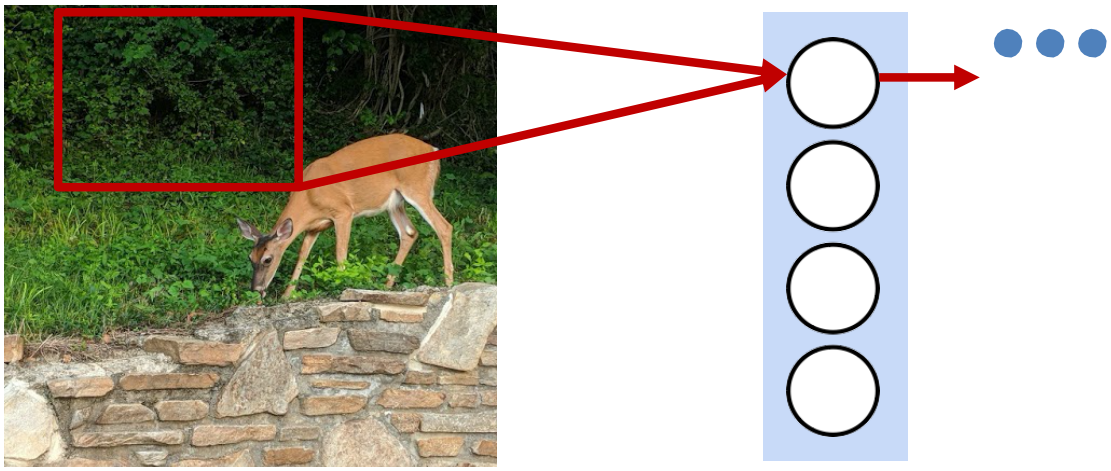
Now

→ Perceptron evaluates this region of the image

→ We can arrange these outputs in form of the original picture

Adapted from Unberath

Scanning: A Closer Look



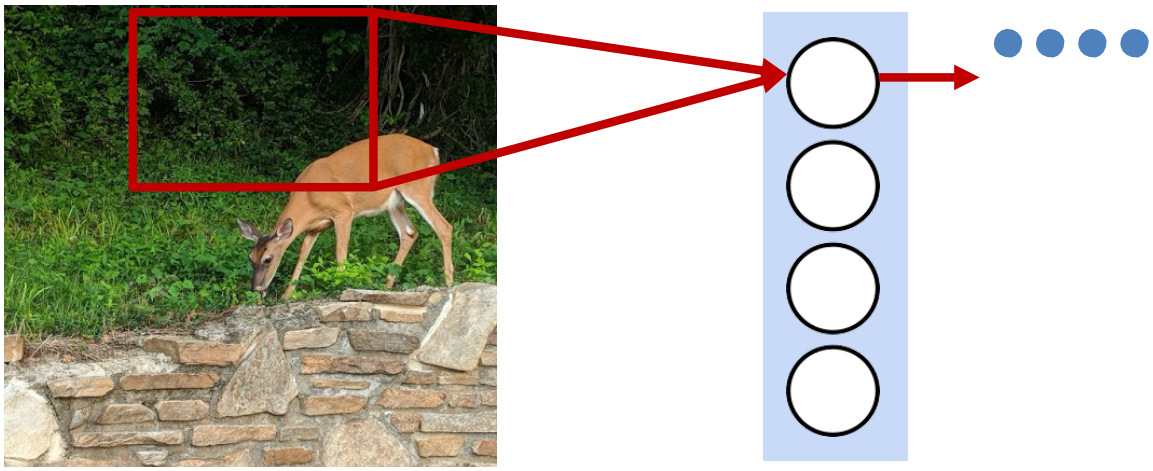
Now

→ Perceptron evaluates this region of the image

→ We can arrange these outputs in form of the original picture

Adapted from Unberath

Scanning: A Closer Look



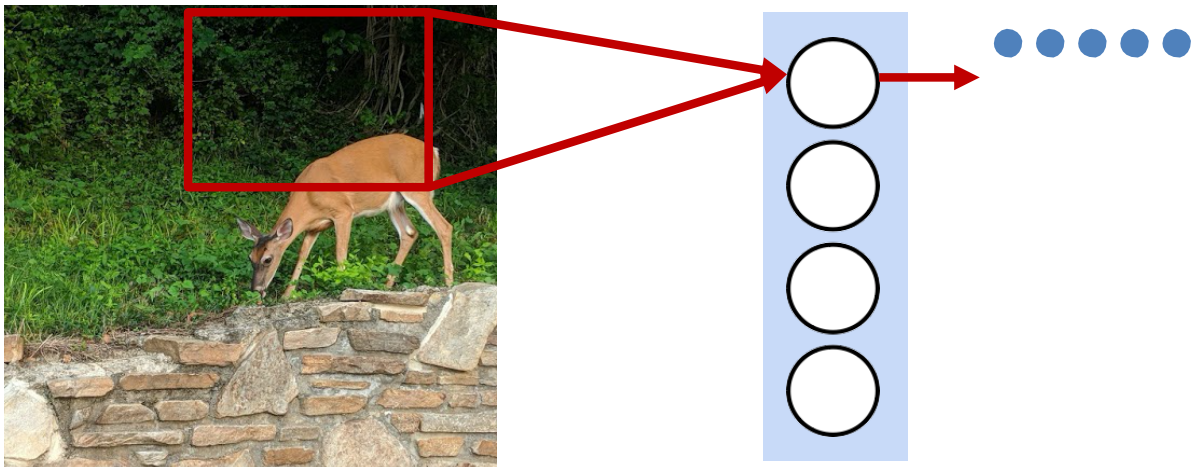
Now

→ Perceptron evaluates this region of the image

→ We can arrange these outputs in form of the original picture

Adapted from Unberath

Scanning: A Closer Look



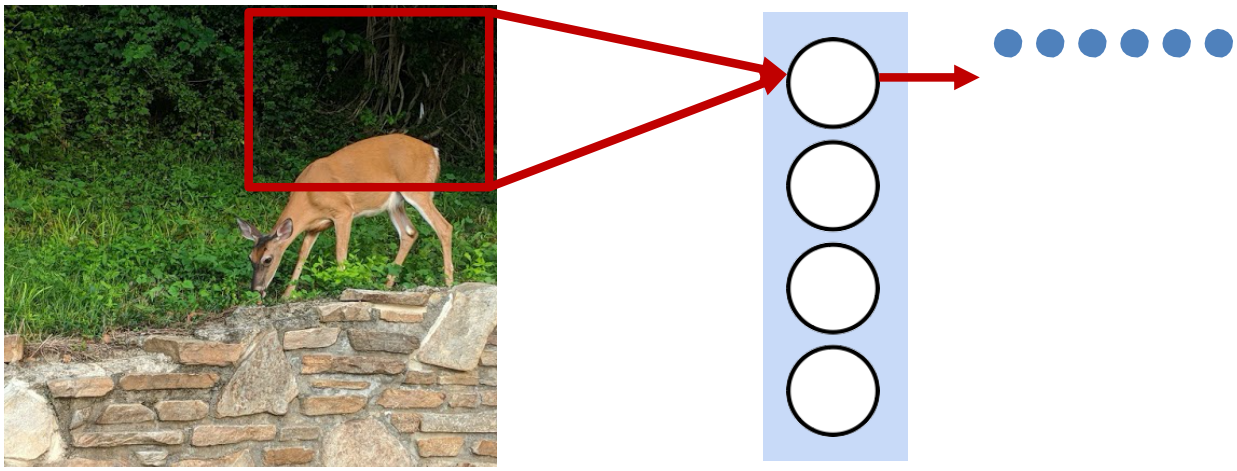
Now

→ Perceptron evaluates this region of the image

→ We can arrange these outputs in form of the original picture

Adapted from Unberath

Scanning: A Closer Look



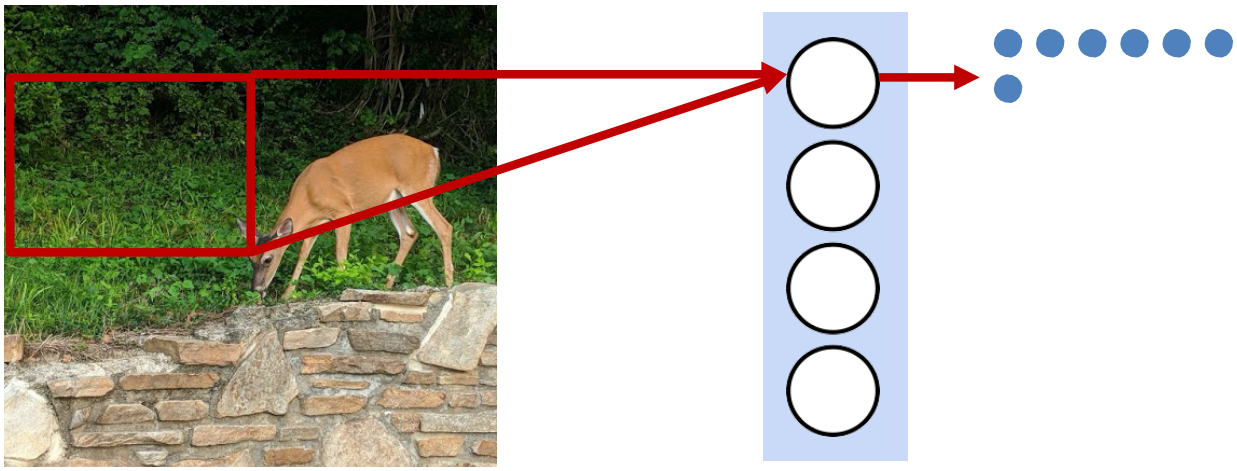
Now

→ Perceptron evaluates this region of the image

→ We can arrange these outputs in form of the original picture

Adapted from Unberath

Scanning: A Closer Look



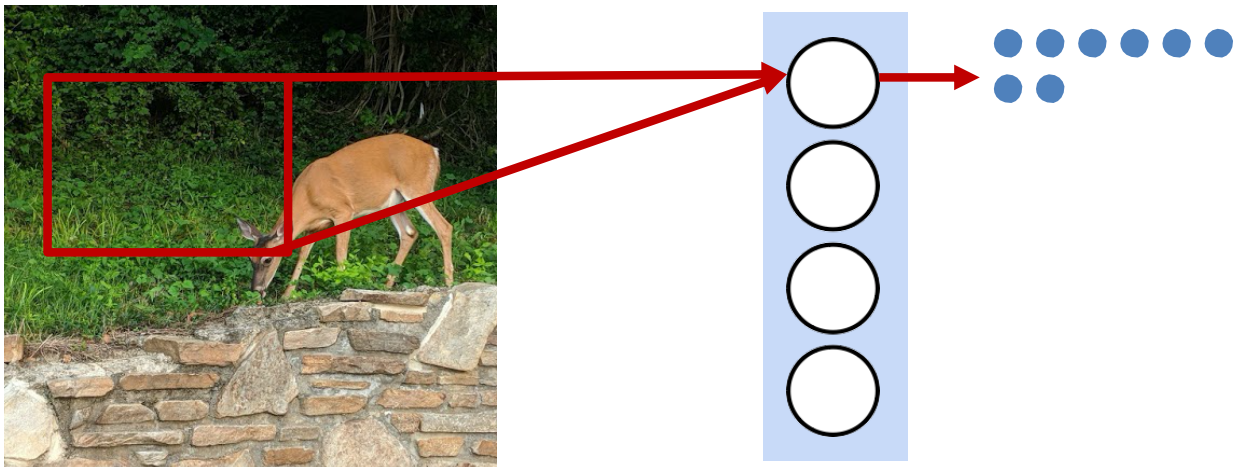
Now

→ Perceptron evaluates this region of the image

→ We can arrange these outputs in form of the original picture

Adapted from Unberath

Scanning: A Closer Look



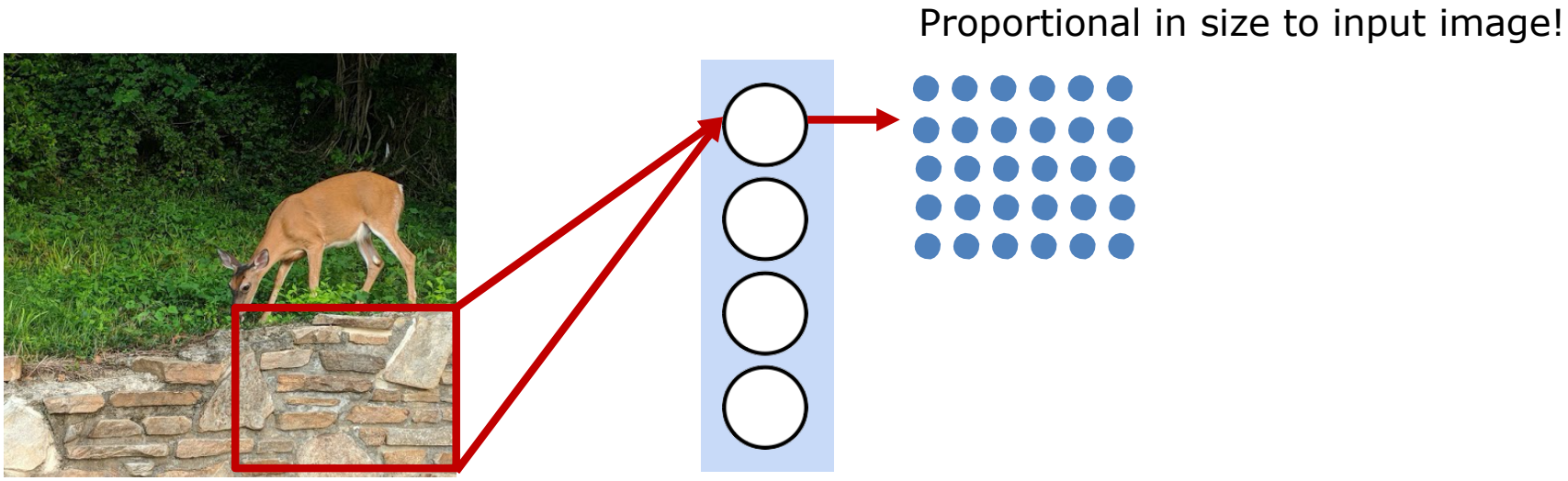
Now

→ Perceptron evaluates this region of the image

→ We can arrange these outputs in form of the original picture

Adapted from Unberath

Scanning: A Closer Look



Now

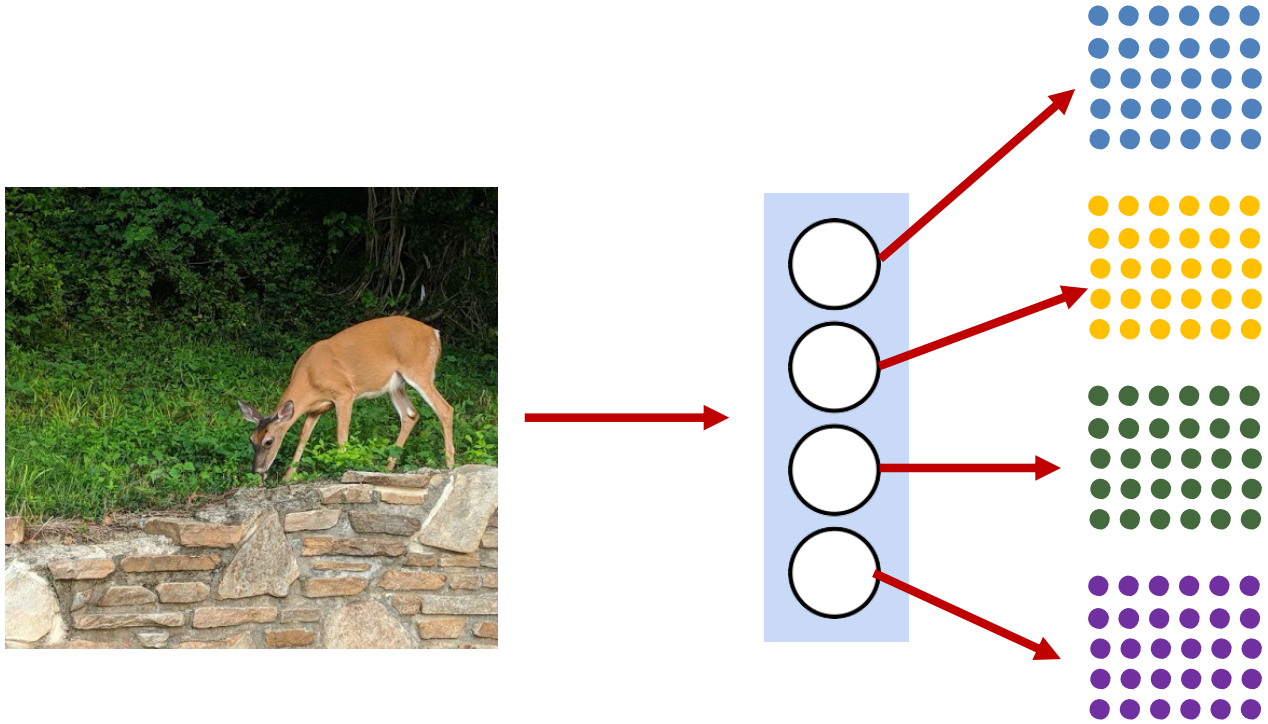
→ Perceptron evaluates this region of the image

→ We can arrange these outputs in form of the original picture

Adapted from Unberath

Scanning: A Closer Look

This can be repeated for every perceptron's output

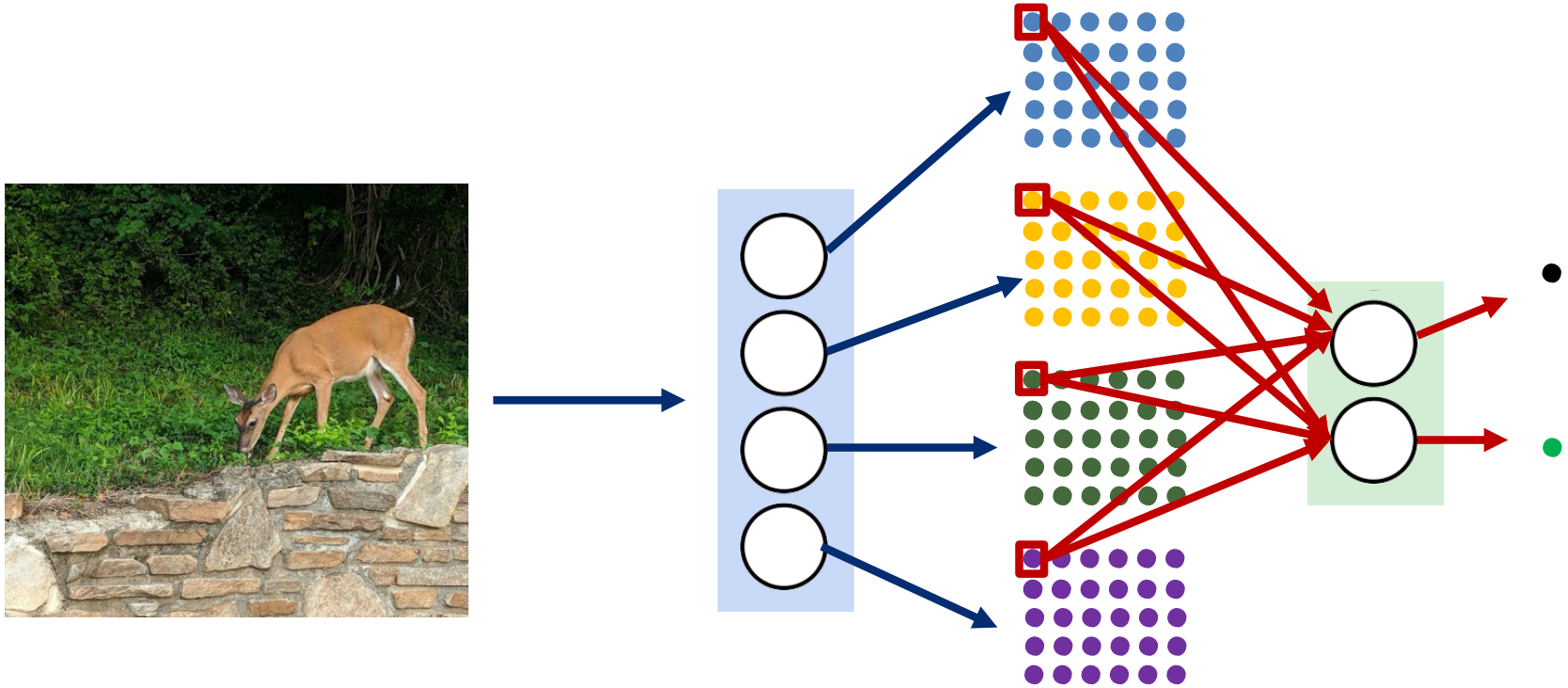


Adapted from Unberath

Scanning: A Closer Look

We can recurse the logic!

Now, perceptrons are jointly scanning multiple “images”

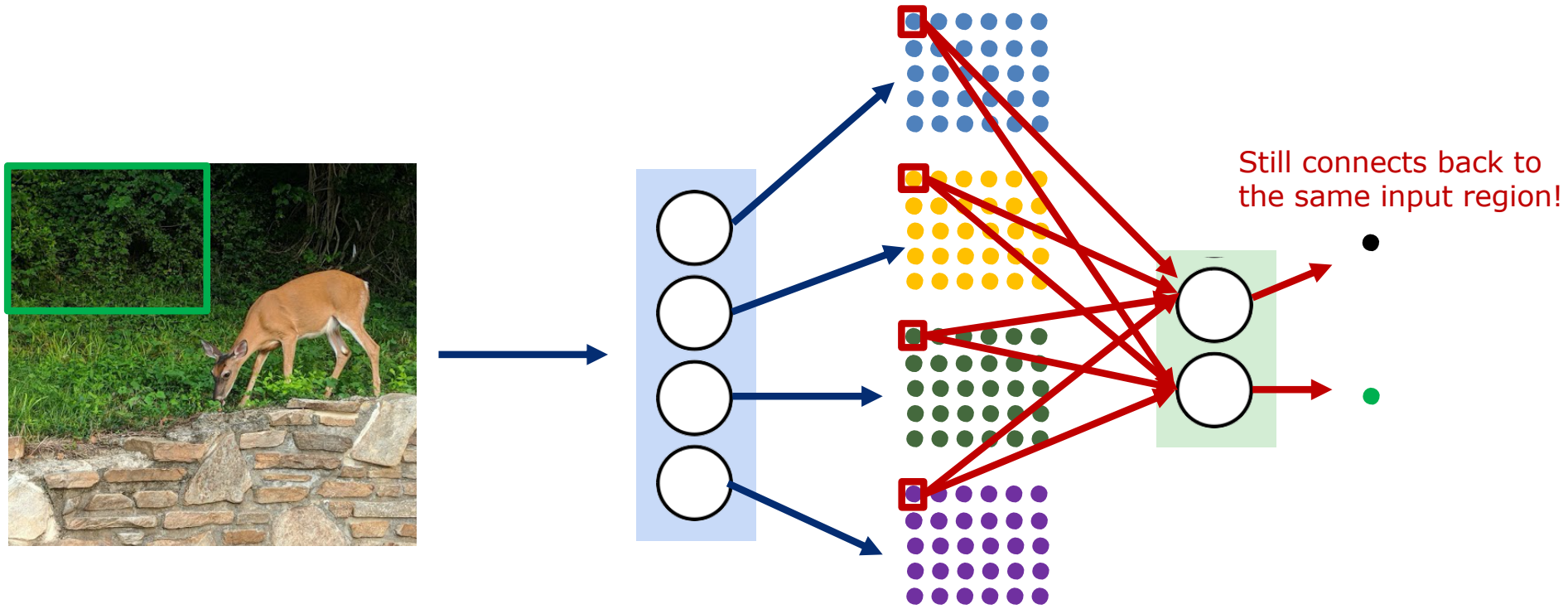


Adapted from Unberath

Scanning: A Closer Look

We can recurse the logic!

Now, perceptrons are jointly scanning multiple “images”

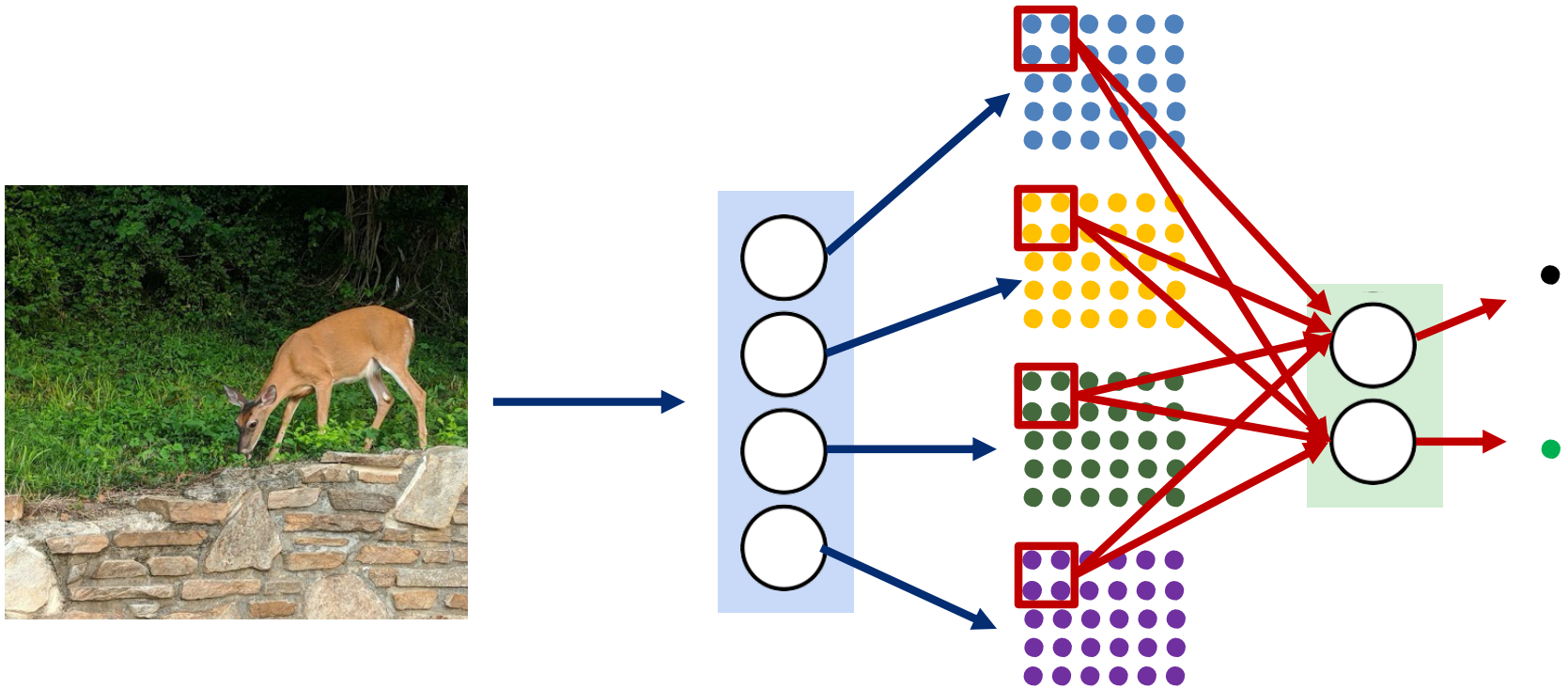


Adapted from Unberath

Scanning: A Closer Look

We can recurse the logic!

Now, perceptrons are jointly scanning multiple “images”

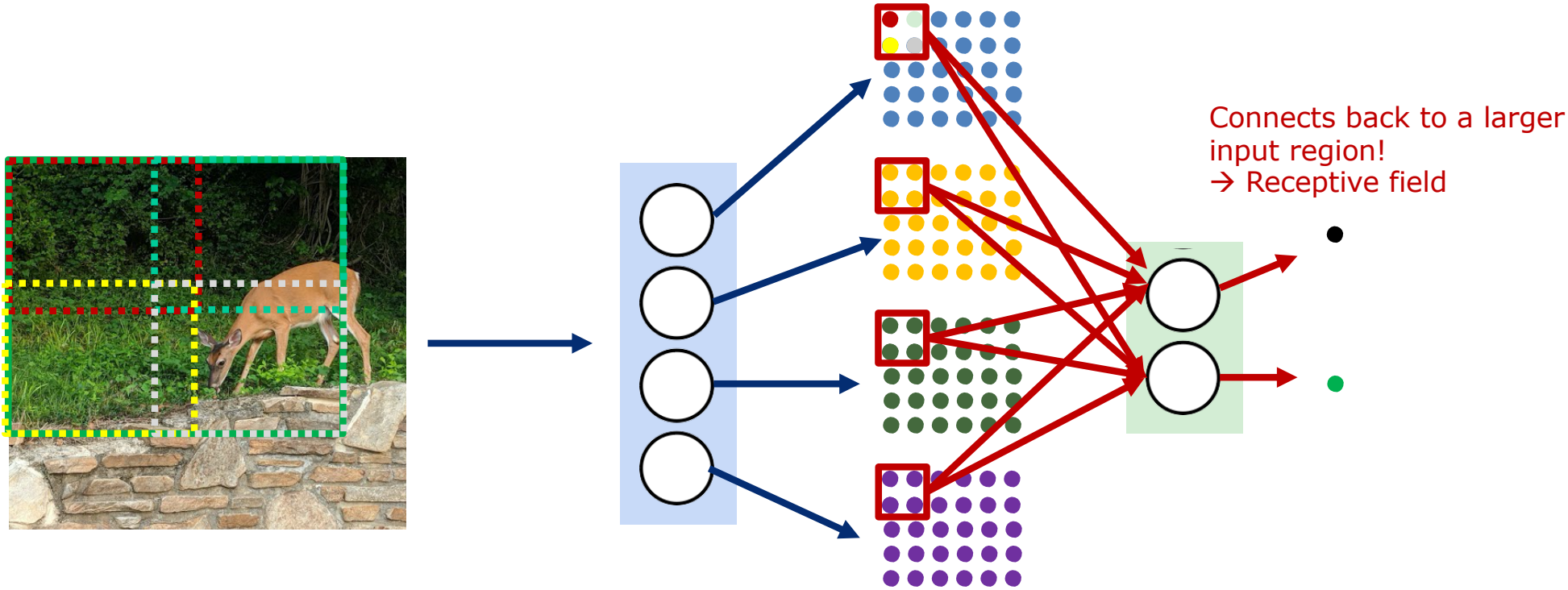


Adapted from Unberath

Scanning: A Closer Look

We can recurse the logic!

Now, perceptrons are jointly scanning multiple “images”

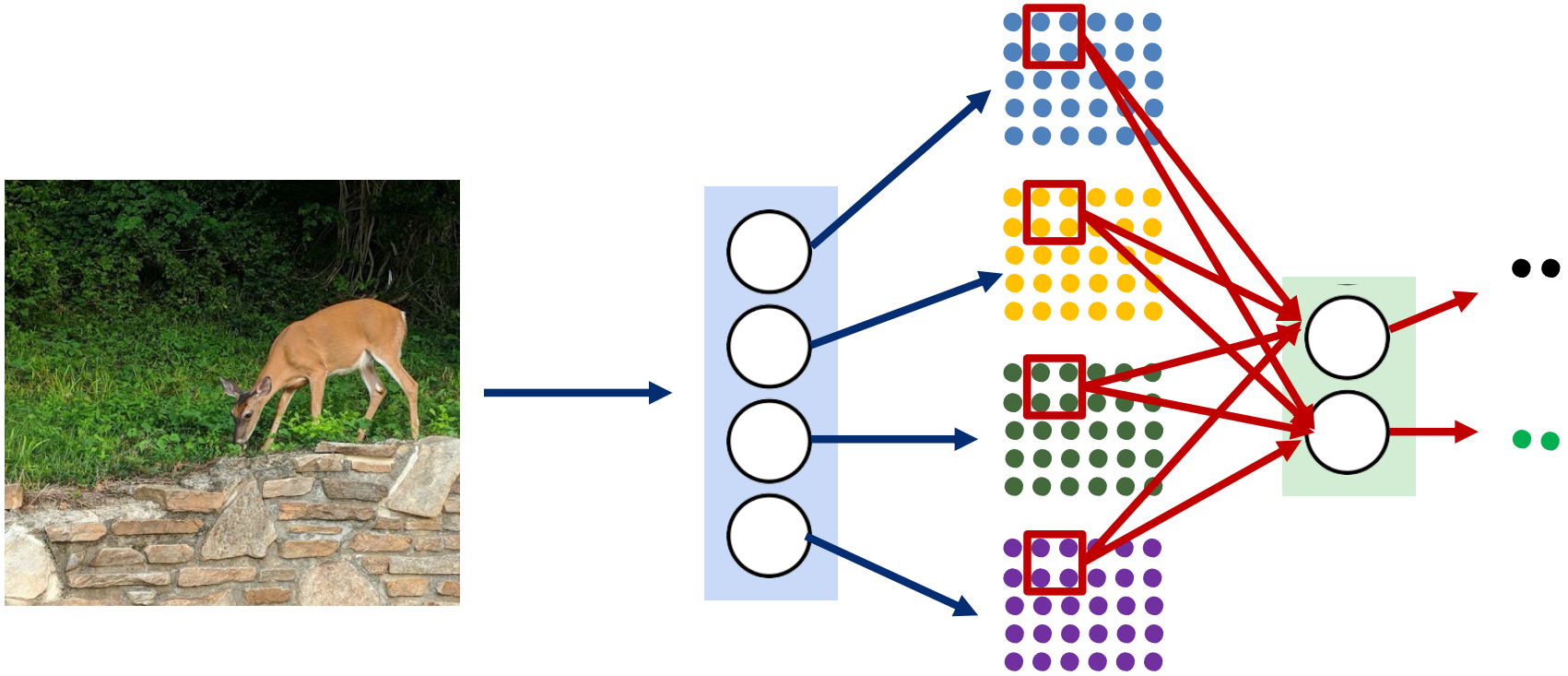


Adapted from Unberath

Scanning: A Closer Look

We can recurse the logic!

Now, perceptrons are jointly scanning multiple “images”

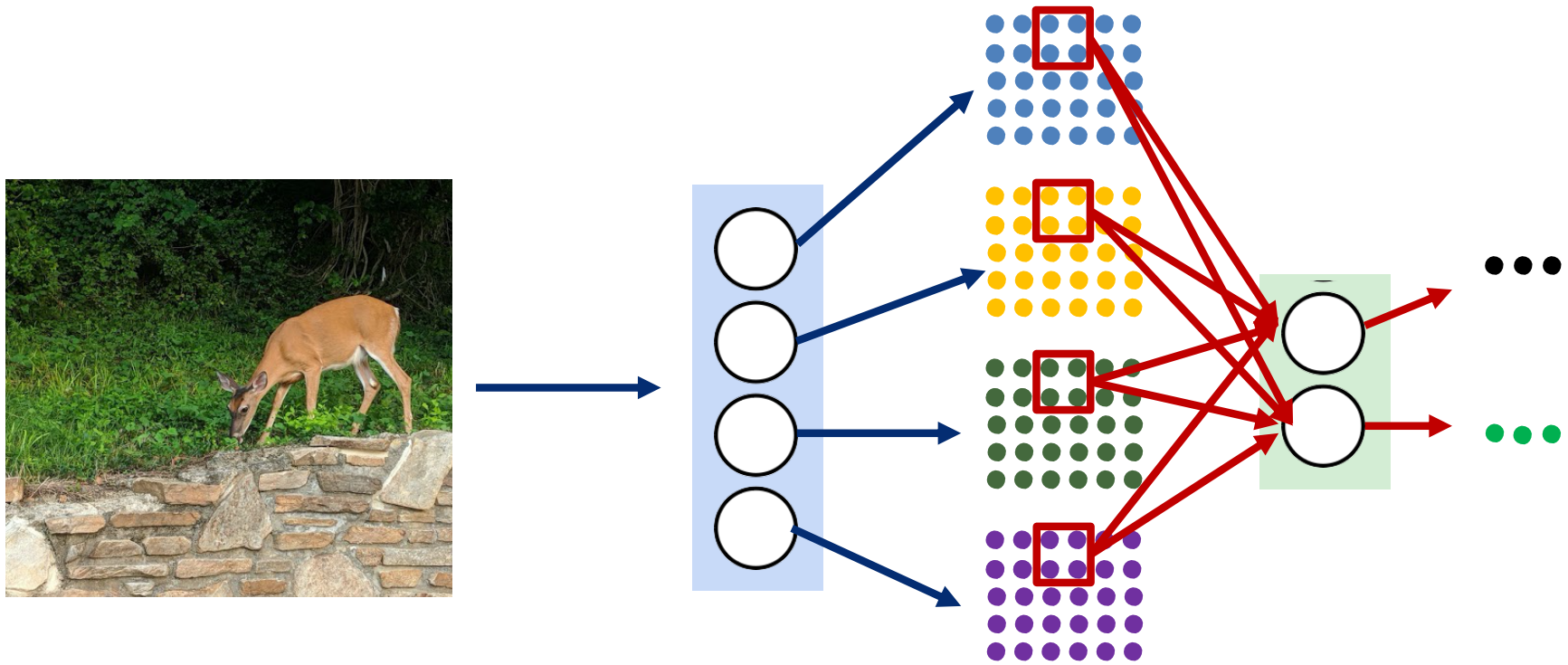


Adapted from Unberath

Scanning: A Closer Look

We can recurse the logic!

Now, perceptrons are jointly scanning multiple “images”

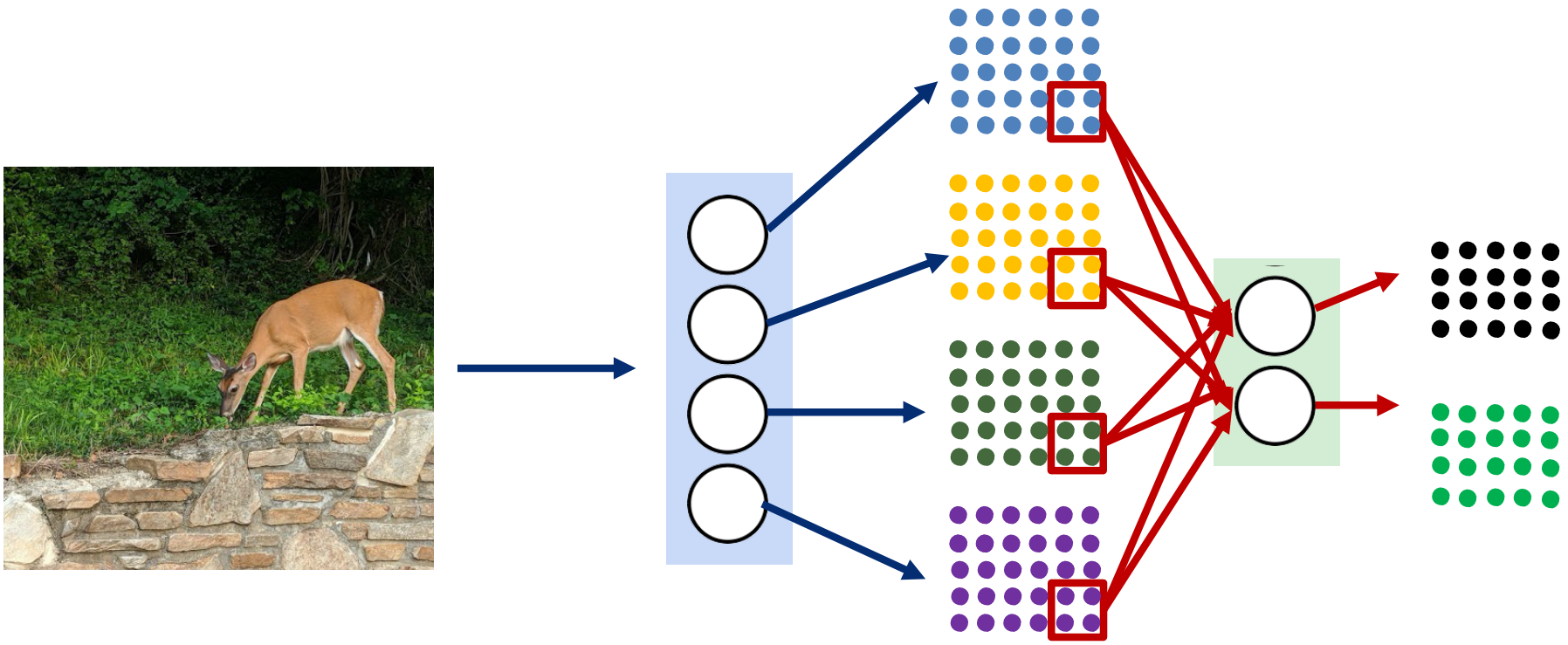


Adapted from Unberath

Scanning: A Closer Look

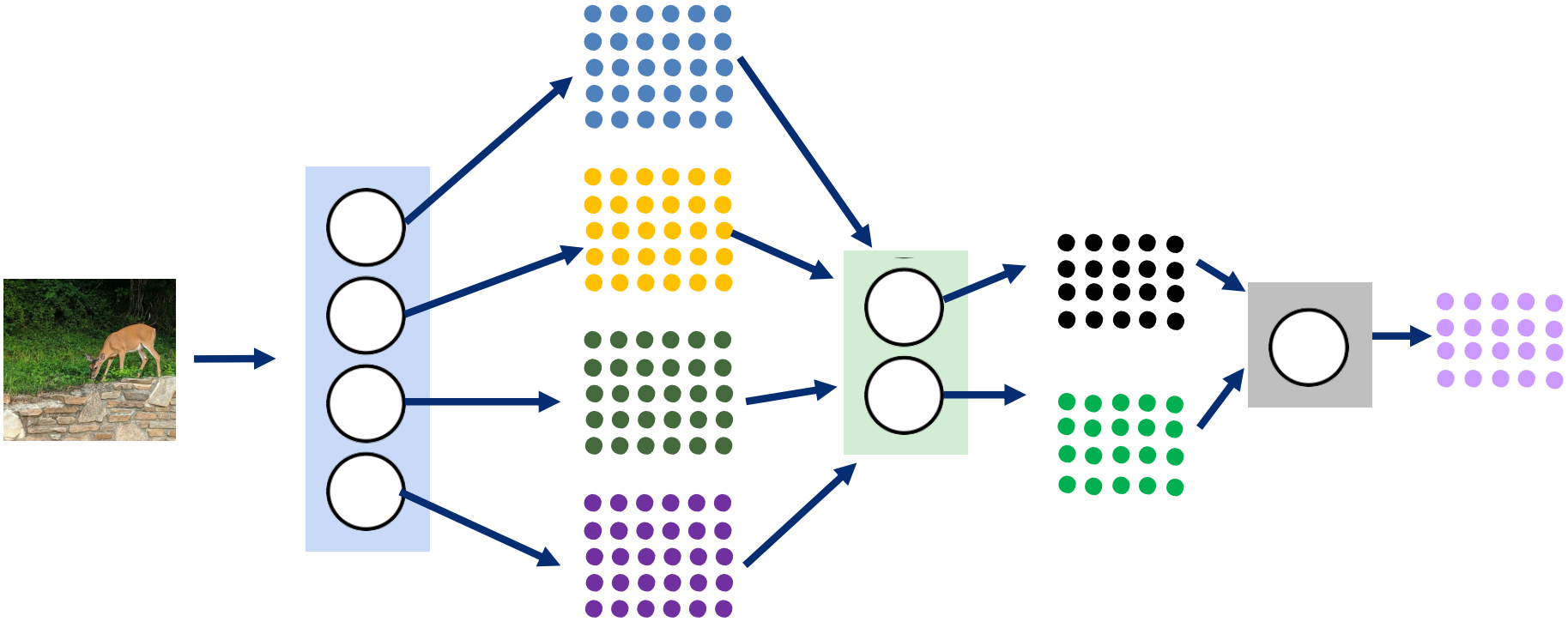
We can recurse the logic!

Now, perceptrons are jointly scanning multiple “images”



Adapted from Unberath

Scanning: A Closer Look

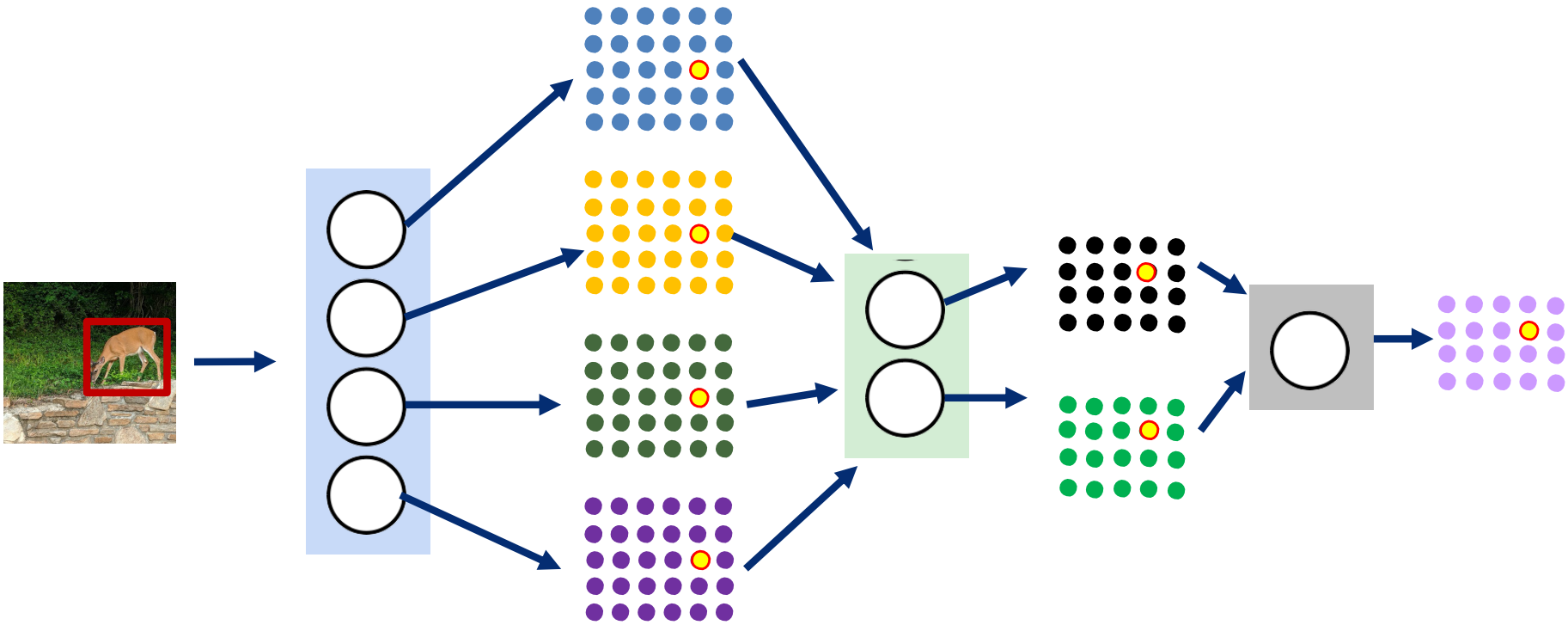


Adapted from Unberath

Scanning: A Closer Look

Output layer must consider last hidden layer!

→ “Detect location of object in image”

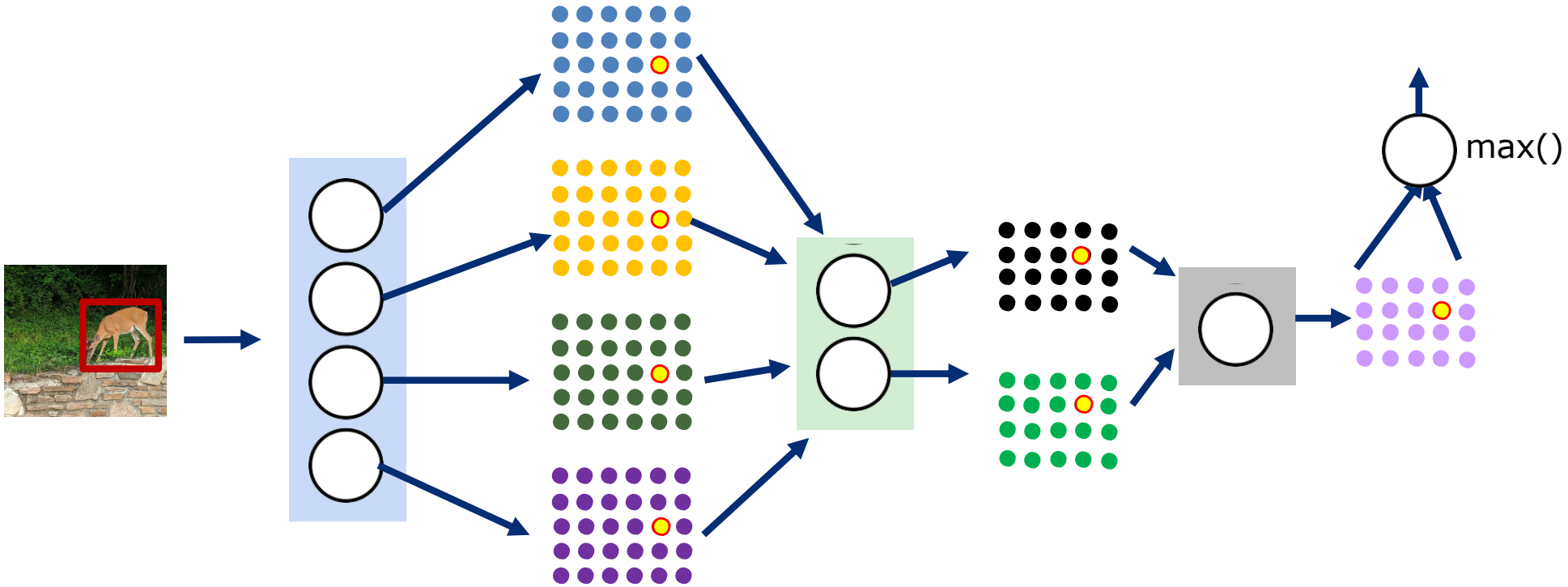


Adapted from Unberath

Scanning: A Closer Look

Output layer must consider last hidden layer!

→ "**Is there such object in image?**"

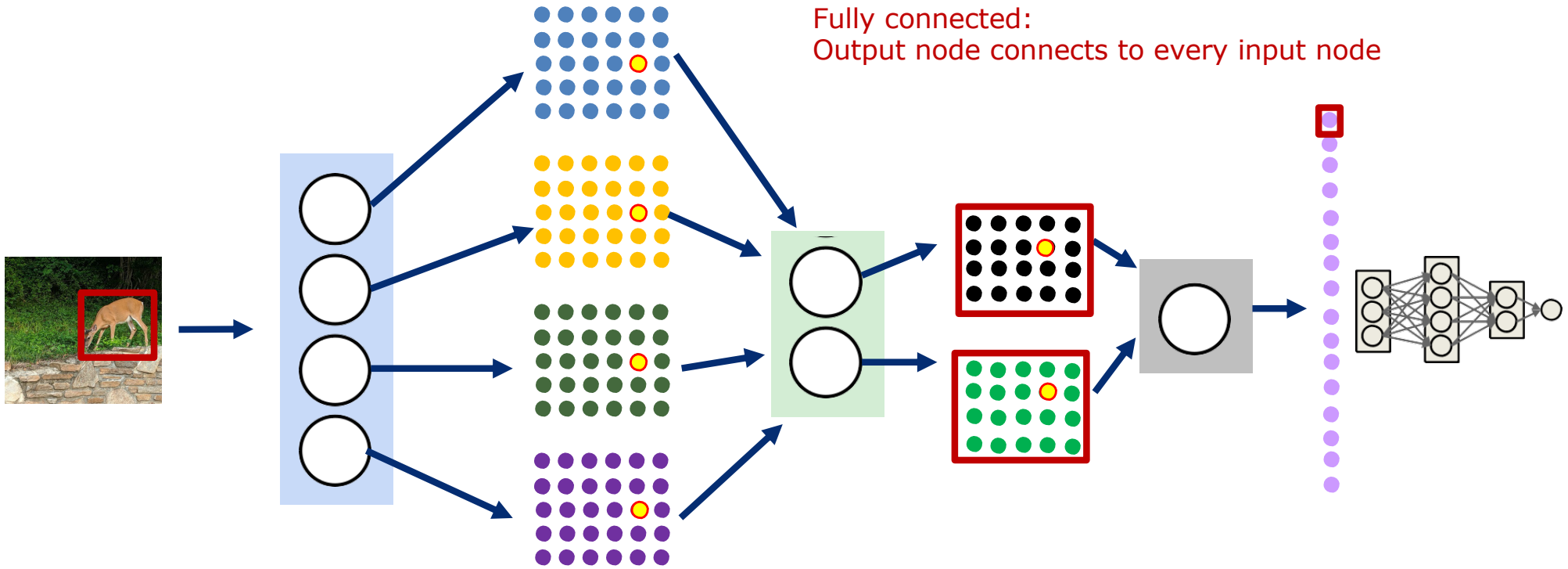


Adapted from Unberath

Scanning: A Closer Look

"**Is there such object in image?**"

→ Flatten output since spatial configuration is no longer important



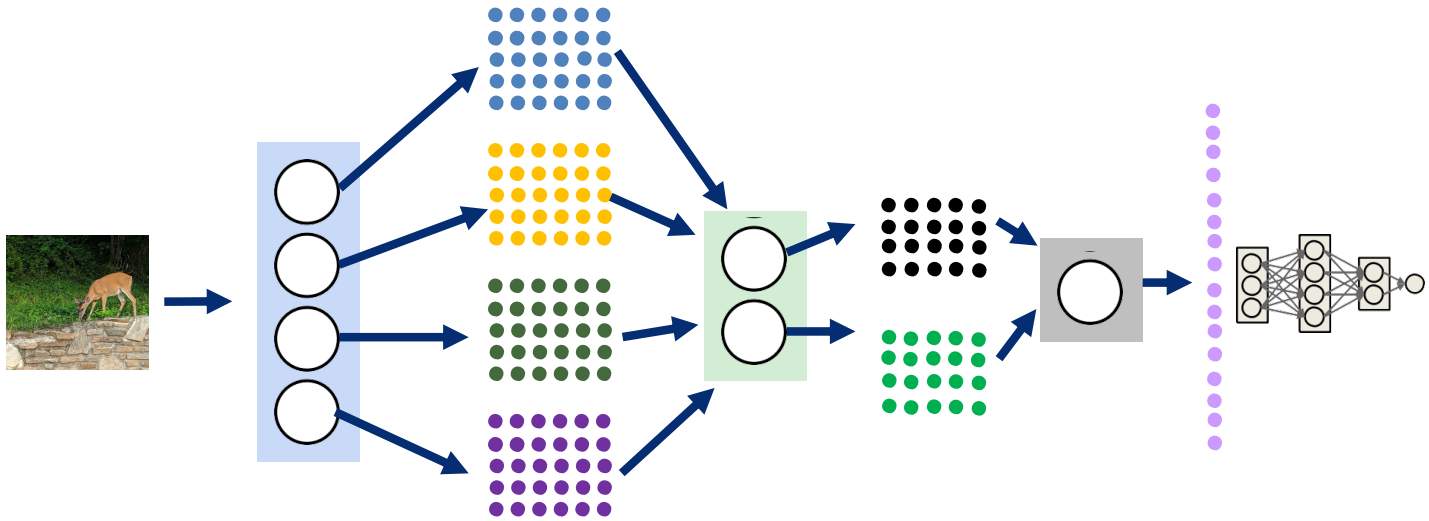
Adapted from Unberath

Hierarchical Build-up of Features

Sharing parameters

→ Distribution forces localized patterns in lower layers (generalizability)

→ Reduction of parameters (because of sharing)



Adapted from Unberath

Recap

- Instead of passing a whole image into a MLP, we can slide a smaller MLP over the image
- This allowed us to recreate an image-like structure in the hidden layer

- We can use different MLPs (with different parameters) for every location
- Or, we can share parameters across the spatial domain
  → Translation invariance!

- Localized features in lower layers
- More abstract, complex features in deeper layers

Adapted from Unberath

A Brief Introduction to Pytorch

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5, 1)
        self.conv2 = nn.Conv2d(20, 50, 5, 1)
        self.fc1 = nn.Linear(4*4*50, 500)
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 4*4*50)
        x = F.relu(self.fc1(x))
        return = self.fc2(x)
```

A Brief Introduction to Pytorch

```
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.cross_entropy(output, target)
        loss.backward()
        optimizer.step()

    if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

A Brief Introduction to Pytorch

```
def test(args, model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.cross_entropy(output, target,
                                         reduction='sum').item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
```

A Brief Introduction to Pytorch

```
train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('../data', train=True, download=True,
                       transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,))])),
         batch_size=args.test_batch_size, shuffle=True, **kwargs)

test_loader = torch.utils.data.DataLoader( datasets.MNIST('../data',
                                train=False, transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,))
                       ])),
        batch_size=args.test_batch_size, shuffle=True, **kwargs)
```

A Brief Introduction to Pytorch

```
model = Net().to(device)
optimizer = optim.SGD(model.parameters(), lr=args.lr,
momentum=args.momentum)

    for epoch in range(1, args.epochs + 1):
        train(args, model, device, train_loader, optimizer, epoch)
        test(args, model, device, test_loader)

    if (args.save_model):
        torch.save(model.state_dict(),"mnist_cnn.pt")
```

References: Textbooks

- Bishop, Christopher M. *Pattern recognition and machine learning*. springer, 2006, Chap 5.

- LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." *nature* 521, no. 7553 (2015): 436-444, Chaps 5-8.