



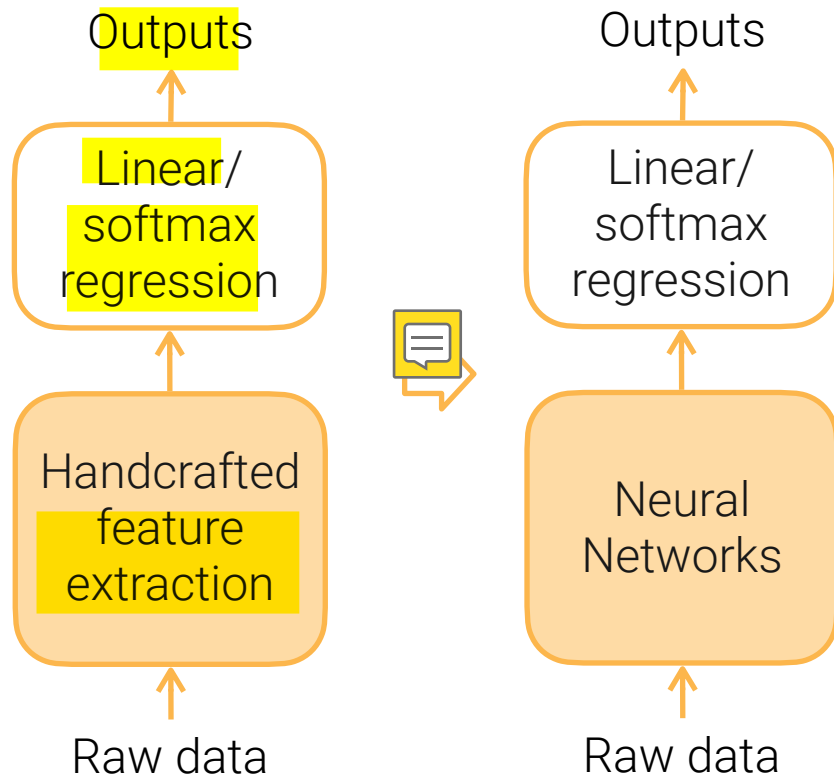
CS 329P : Practical Machine Learning (2021 Fall)

3.4 Neural Networks

Qingqing Huang, Mu Li, Alex Smola

<https://c.d2l.ai/stanford-cs329p>

Handcrafted Features → Learned Features



- NN usually requires more data and more computation
- NN architectures to model data structures
 - Multilayer perceptions
 - Convolutional neural networks
 - Recurrent neural networks
 - Attention mechanism
- Design NN to incorporate prior knowledge about the data

Linear Methods → Multilayer Perceptron (MLP)



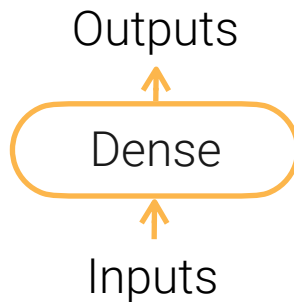
- A **dense** (fully connected, or linear) layer has parameters

$\mathbf{W} \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^m$, it computes output $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \in \mathbb{R}^m$

- Linear regression: dense layer with 1 output
- Softmax regression:
dense layer with m outputs + softmax



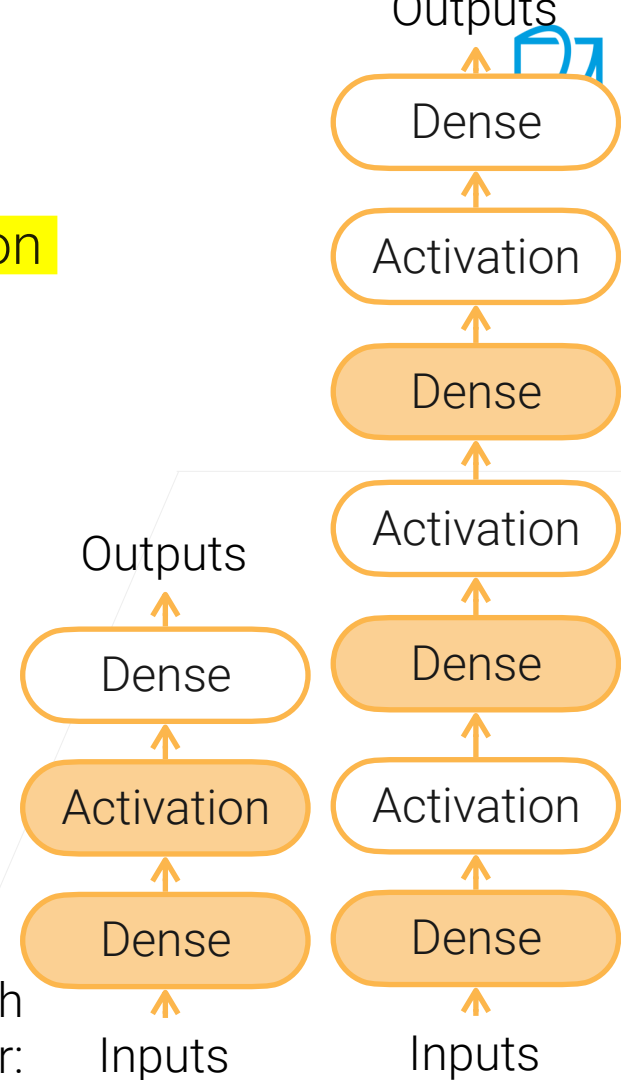
Linear
methods:



Multilayer Perceptron (MLP)



- Activation is an elemental-wise non-linear function
 - $\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$, $\text{ReLU}(x) = \max(x, 0)$
 - It leads to non-linear models
- Stack multiple hidden layers (dense + activation) to get deeper models
- Hyper-parameters:
 - # hidden layers, # outputs of each hidden layer
- Universal approximation theorem



Code



- MLP with 1 hidden layer
- Hyperparameter: num_hiddens

```
def relu(X):  
    return torch.max(X, 0)
```

```
W1 = nn.Parameter(torch.randn(num_inputs, num_hiddens) * 0.01)  
b1 = nn.Parameter(torch.zeros(num_hiddens))  
W2 = nn.Parameter(torch.randn(num_hiddens, num_outputs) * 0.01)  
b2 = nn.Parameter(torch.zeros(num_outputs))
```

```
H = relu(X @ W1 + b1)  
Y = H @ W2 + b2
```

Full code: http://d2l.ai/chapter_multilayer-perceptrons/mlp-scratch.html

Dense layer → Convolution layer



- Learn ImageNet (300x300 images with 1K classes) by a MLP with a single hidden layer with 10K outputs
 - It leads to 1 billion learnable parameters, that's too big!
 - Fully connected: an output is a weighted sum over all inputs
- Recognize objects in images
 - Translation invariance: similar output no matter where the object is
 - Locality: pixels are more related to near neighbors
- Build the prior knowledge into the model structure
 - Achieve same model capacity with less # params



Convolution layer

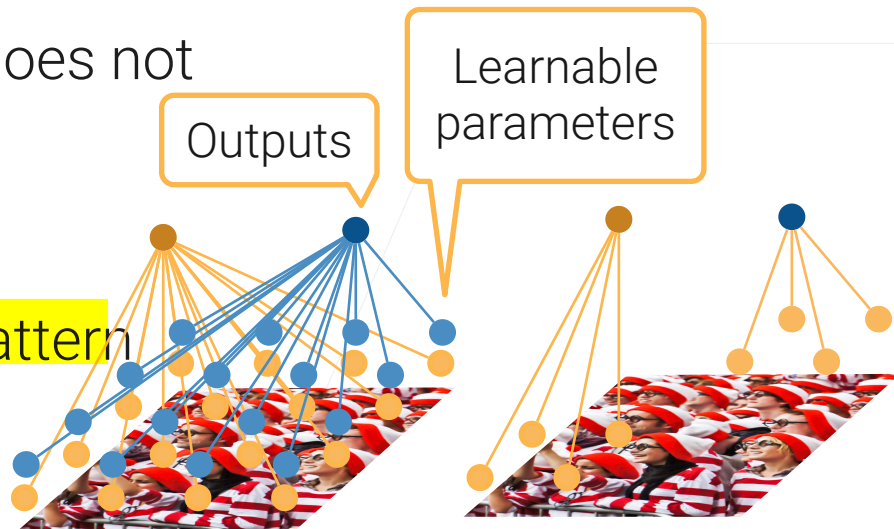


- Locality: an output is computed from $k \times k$ input windows
- Translation invariant: outputs use the same $k \times k$ weights (kernel)
- # model params of a conv layer does not depend on input/output sizes

$$n \times m \rightarrow k \times k$$



- A kernel may learn to identify a pattern



Dense Layer

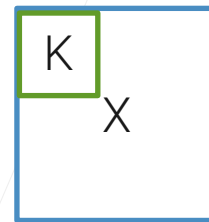
Convolution

Code

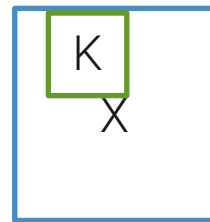


- Convolution with matrix input and matrix output (single channel)

```
# both input `X` and weight `K` are matrices
h, w = K.shape
Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
# stride = 1
for i in range(Y.shape[0]):
    for j in range(Y.shape[1]):
        Y[i, j] = (X[i : i+h, j : j+w] * K).sum()
```



Y[0, 0]




Y[0, 1]

- Exercise: implement multi-channel input / output convolution

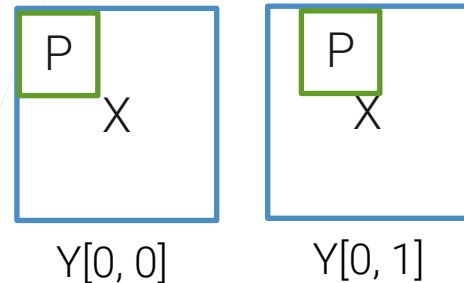
Full code: http://d2l.ai/chapter_convolutional-neural-networks/conv-layer.html

Pooling Layer



- Convolution is sensitive to location 
- A translation/rotation of a pattern in the input results in similar changes of a pattern in the output
- A pooling layer computes mean/max in windows of size $k \times k$

```
# h, w: pooling window height and width
# mode: max or avg
Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
for i in range(Y.shape[0]):
    for j in range(Y.shape[1]):
        if mode == 'max':
            Y[i, j] = X[i : i+h, j : j+w].max()
        elif mode == 'avg':
            Y[i, j] = X[i : i+h, j : j+w].mean()
```



Full code: http://d2l.ai/chapter_convolutional-neural-networks/pooling.html

Convolutional Neural Networks (CNN)

- Stacking convolution layers to extract features
 - Activation is applied after each convolution layer
 - Using pooling to reduce location sensitivity
- Modern CNNs are deep neural network with various hyper-parameters and layer connections (AlexNet, VGG, Inceptions, ResNet, MobileNet)



Outputs

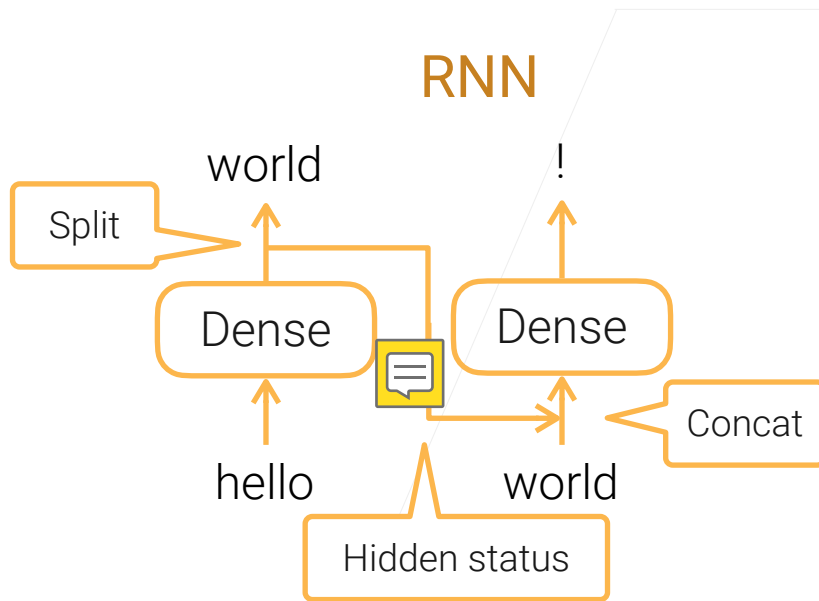
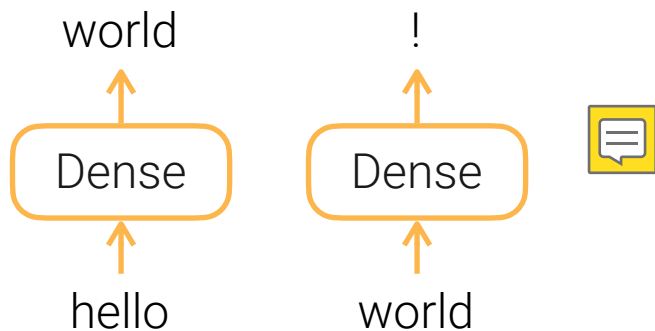
```
lenet = nn.Sequential(  
    nn.Conv2d(...),  
    nn.Sigmoid(),  
    nn.AvgPool2d(...),  
    nn.Conv2d(),  
    nn.Sigmoid(),  
    nn.AvgPool2d(...),  
    nn.Flatten(),  
    nn.Linear(...),  
    nn.Sigmoid(),  
    nn.Linear(...),  
    nn.Sigmoid(),  
    nn.Linear(...))
```



Dense layer → Recurrent networks



- Language model: predict the next word
 - hello → world hello world → !
- Use MLP naively doesn't handle sequence info well:

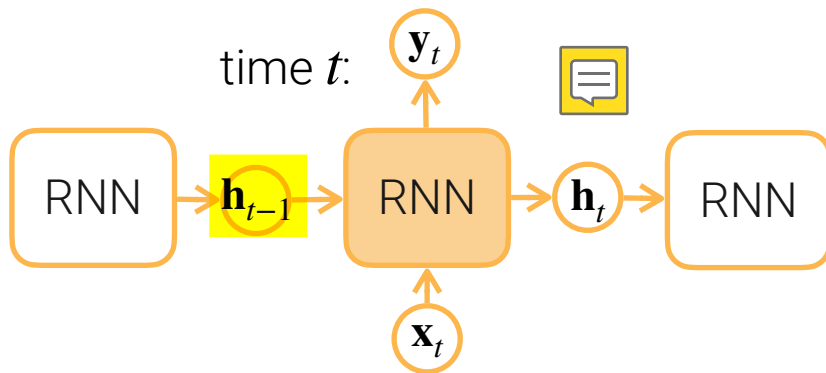


RNN and Gated RNN



The only diff to MLP

- Simple RNN: $\mathbf{h}_t = \phi(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{b}_h)$



- Gated RNN (LSTM, GRU): finer control of information flow
 - Forget input: suppress \mathbf{x}_t when computing \mathbf{h}_t
 - Forget past: suppress \mathbf{h}_{t-1} when computing \mathbf{h}_t



Code



- Implement Simple RNN

```
W_xh = nn.Parameter(torch.randn(num_inputs, num_hiddens) * 0.01)
W_hh = nn.Parameter(torch.randn(num_hiddens, num_hiddens) * 0.01)
b_h = nn.Parameter(torch.zeros(num_hiddens))

H = torch.zeros(num_hiddens)
outputs = []

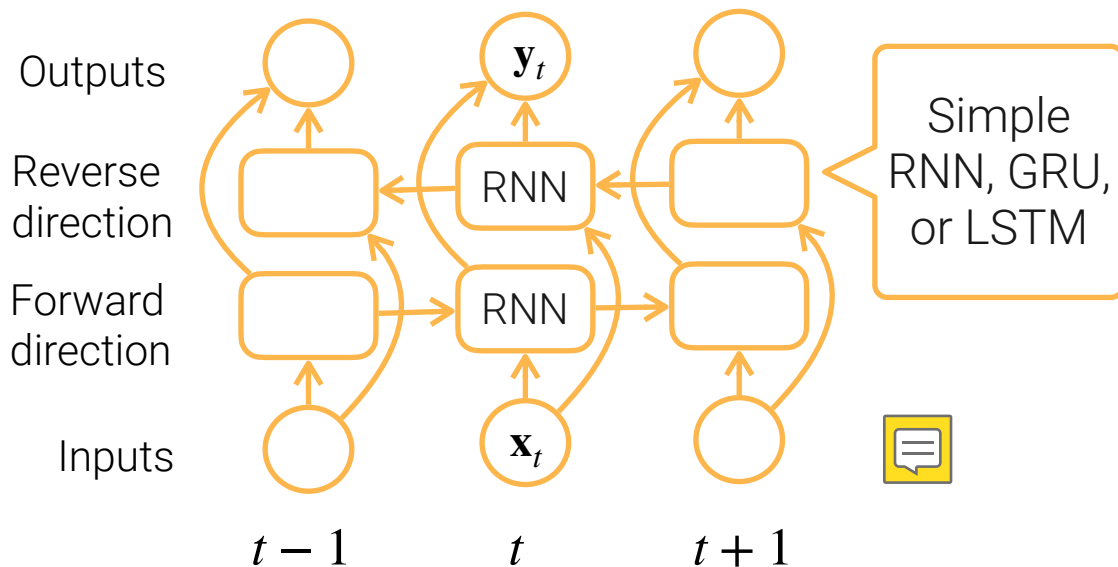
for X in inputs: # `inputs` shape : (num_steps, batch_size, num_inputs)
    H = torch.tanh(X @ W_xh + H @ W_hh + b_h)
    outputs.append(H)
```

Full code at http://d2l.ai/chapter_recurrent-neural-networks/rnn-scratch.html

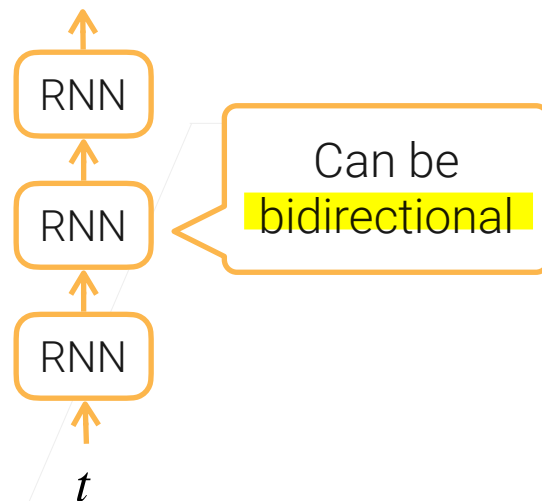
Bi-RNN and Deep RNN



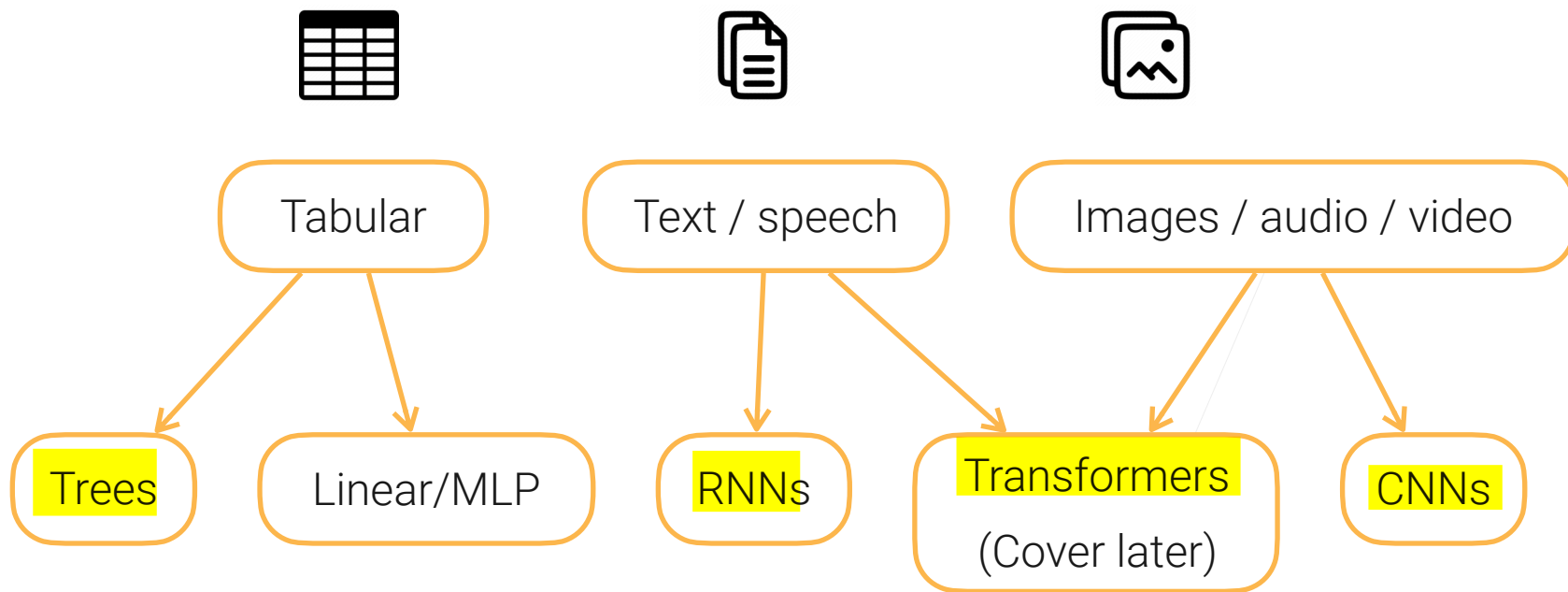
Bidirectional RNN



Deep RNN




Model Selections



Summary



- MLP: stack dense layers with non-linear activations
- CNN: stack convolution activation and pooling layers to efficiently extract spatial information 
- RNN: stack recurrent layers to pass temporal information through hidden state