

Deep learning

Feed-forward and convolutional neural networks for image recognition

Qixiang Fang

Based on Erik-Jan van Kesteren's slides

Last weeks

- Dimension reduction (PCA, SVD, etc.)
- Principal components regression, partial least squares
- Clustering
- Gaussian mixture models (Mclust)

Today

- Introduction to neural networks
- Feed-forward neural networks
- Estimation / optimization
- Convolutional neural networks
- Battling the curse of dimensionality

Learning goal: getting started with & generating understanding about neural networks

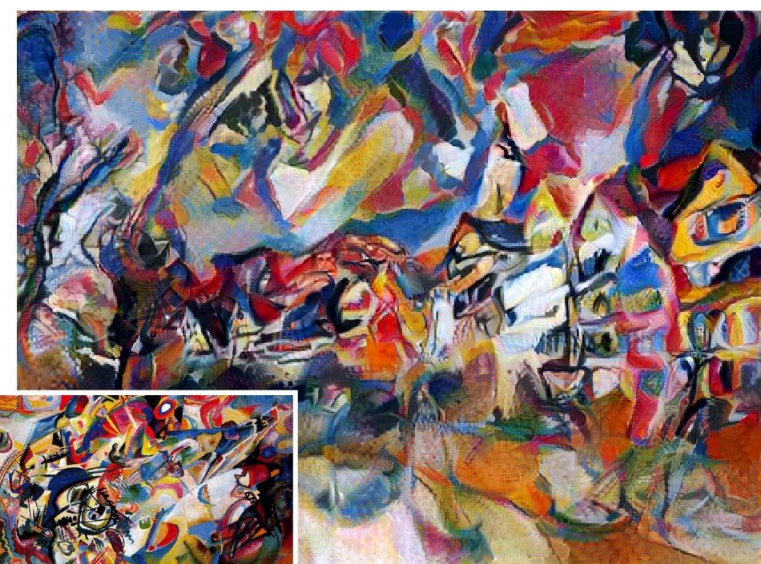
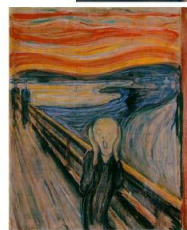
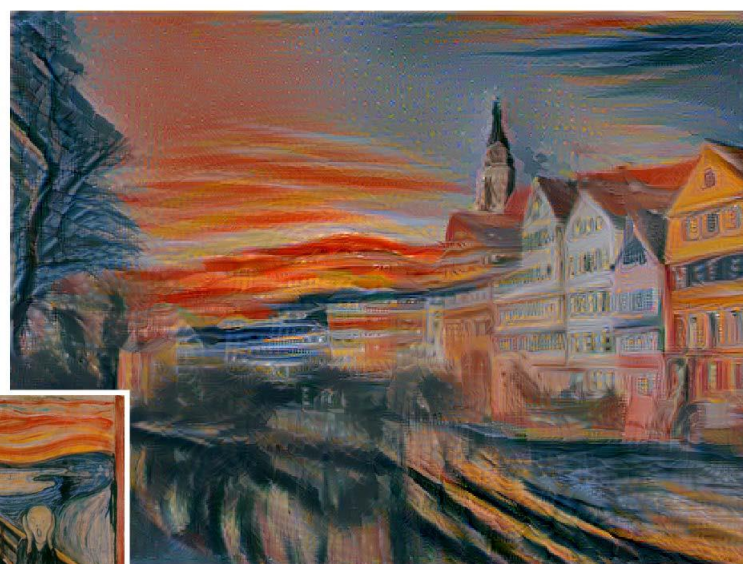
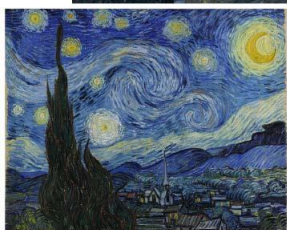
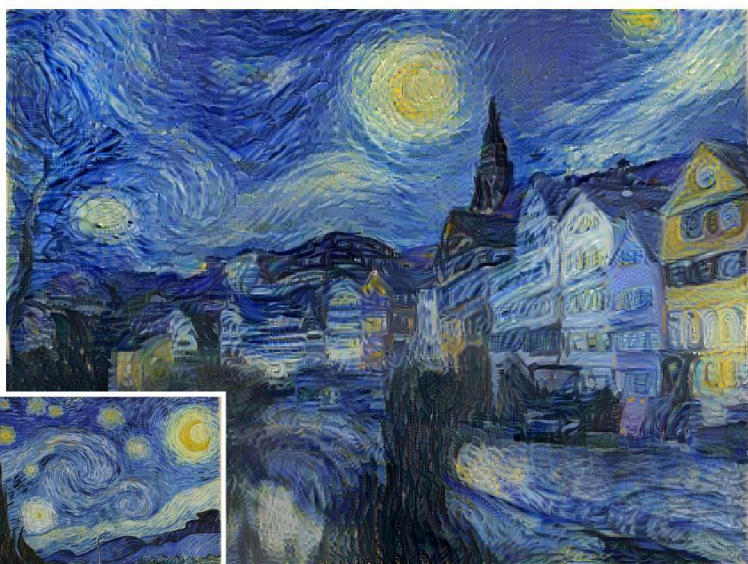
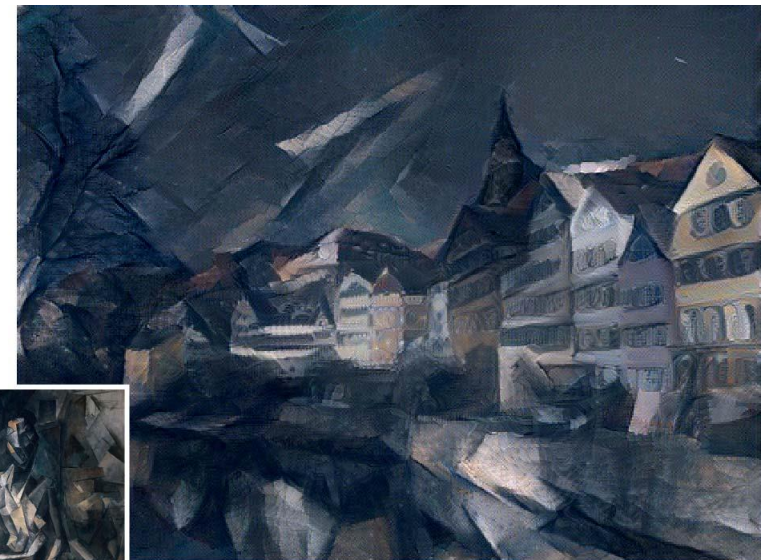
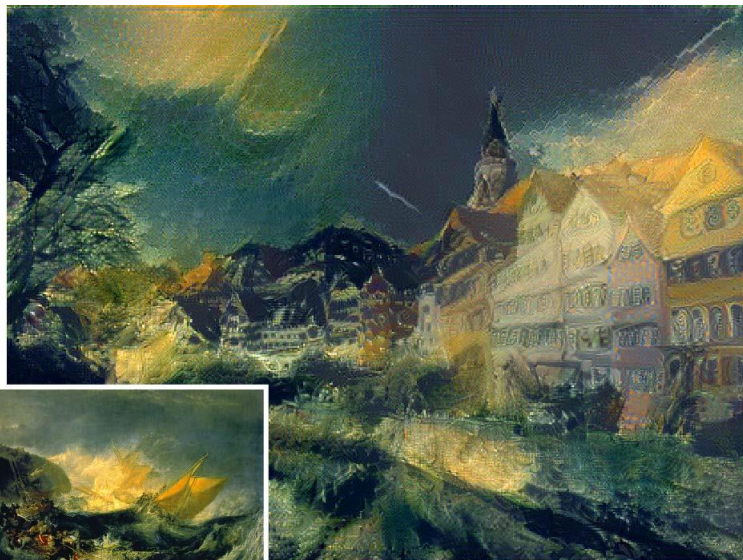
Introduction

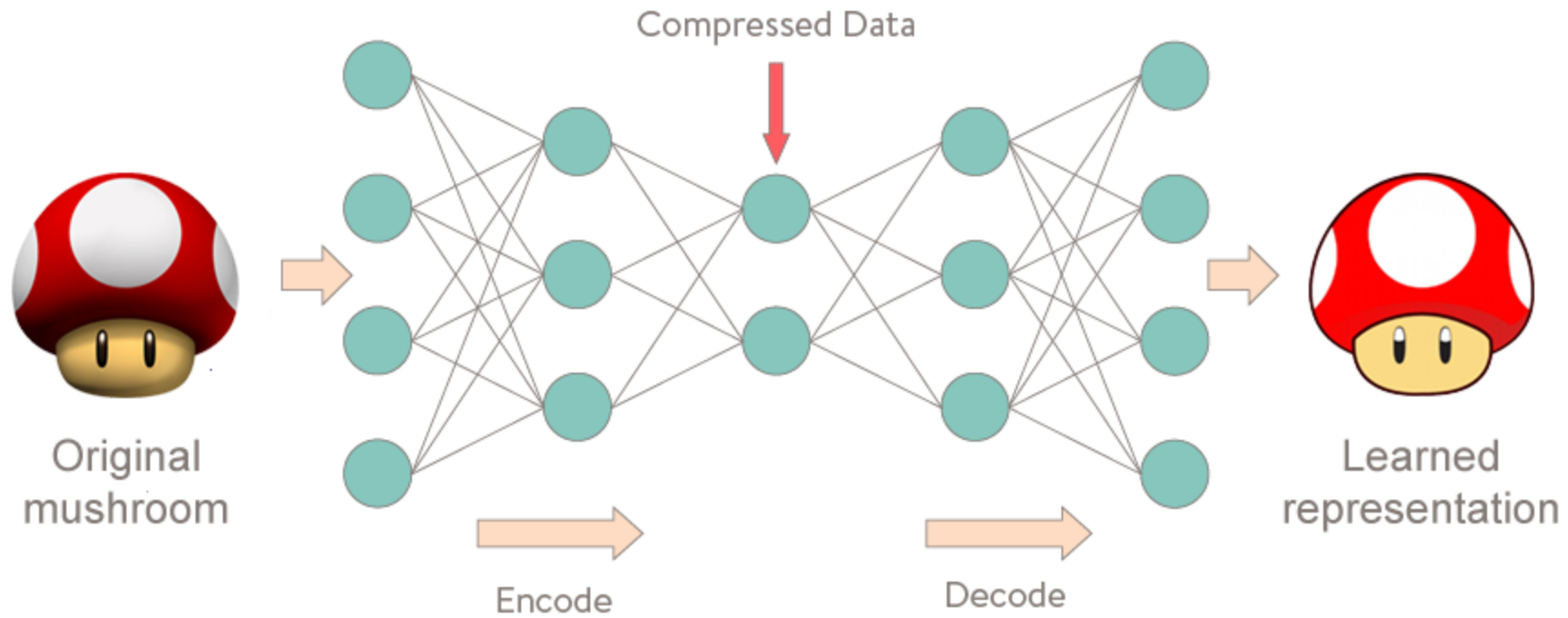
Why should we learn this?

State-of-the-art performance on various tasks

- Text generation (ChatGPT)
- Text mining and Natural Language Processing (next two weeks!)
- Weather forecasting
- Object recognition
- Spam filtering
- Image generation
- Style transfer
- ...

<https://thispersondoesnotexist.com/>





<https://community.canvaslms.com/t5/Canvas-Developers-Group/Canvas-LMS-Cheat-Detection-System-In-Python/m-p/118134>

So what is a neural network?

Neural networks

$$y = f(X) + \epsilon$$

- Neural networks are a way to specify $f(X)$
- You can display $f(X)$ graphically
- Let's graphically represent linear regression!

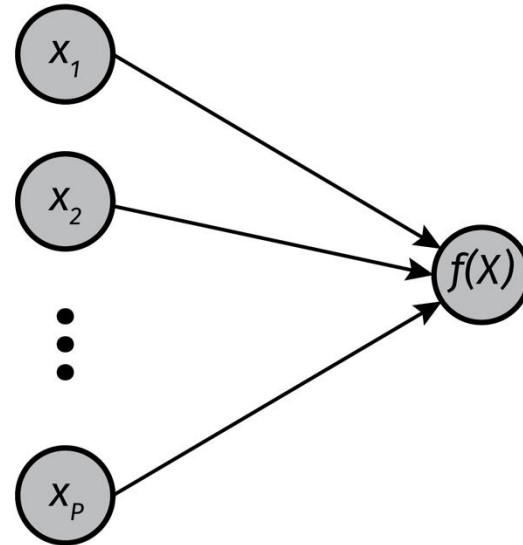
$$f(X_i) = \sum_{p=1}^P \beta_p x_{pi}$$

Linear regression as neural net

Graphical representation

- Parameters are arrows
- Arrows ending in a node are summed together
- Intercept is not drawn

$$f(X_i) = \alpha + \sum_{p=1}^P \beta_p x_{pi}$$

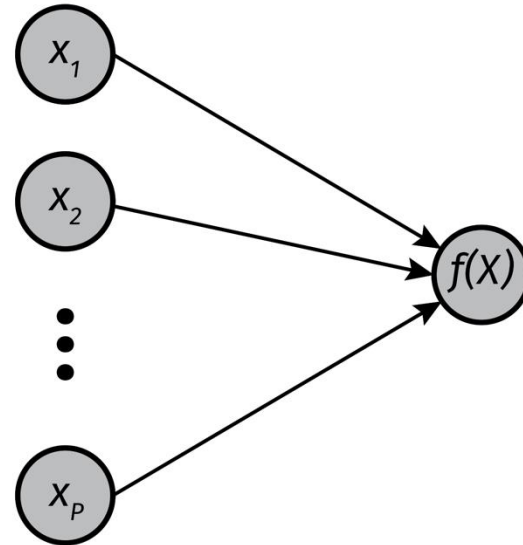


Linear regression as neural net

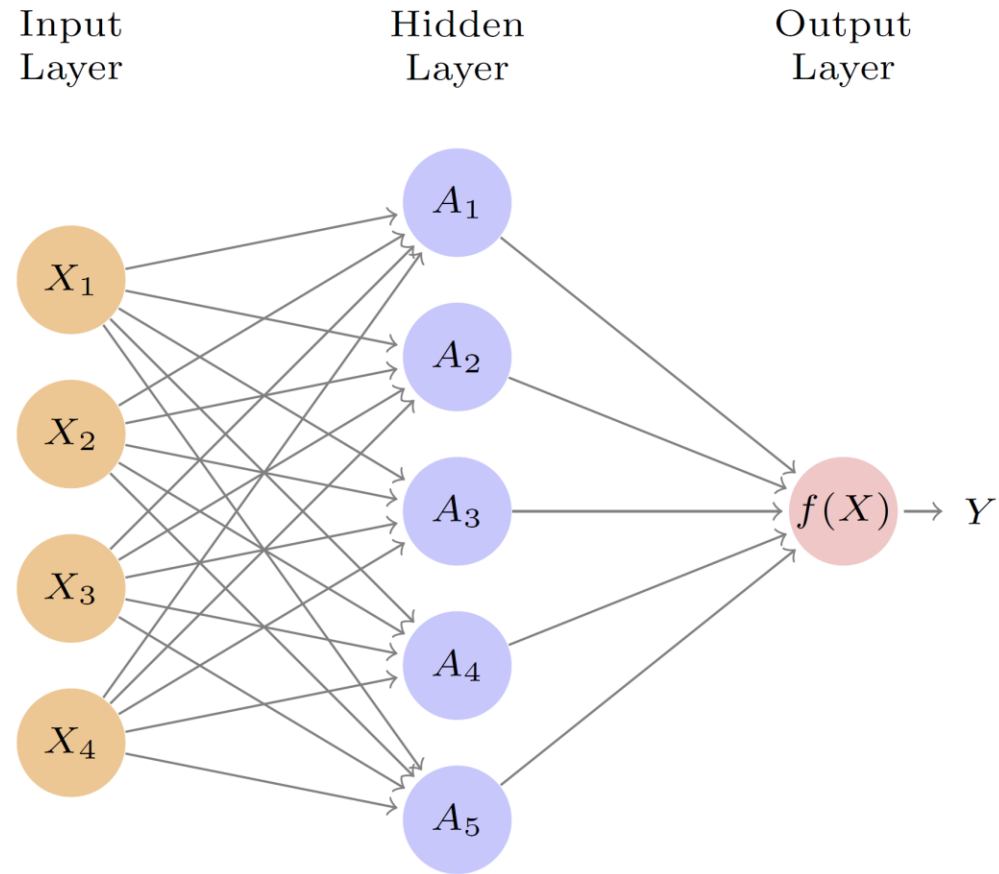
Neural network jargon

- Parameter = **weight**
- Intercept = **bias**

$$f(X_i) = \beta + \sum_{p=1}^P w_p x_{pi}$$



Single layer neural networks



Single layer neural networks

$$y = f(X) + \epsilon$$

Specify a layer with K *hidden units* called A

$$f(X) = \beta_0 + \sum_{k=1}^K \beta_k A_k$$

Where

$$A_k = h_k(X) = g \left(\beta_k + \sum_{p=1}^P w_{pk} x_p \right)$$

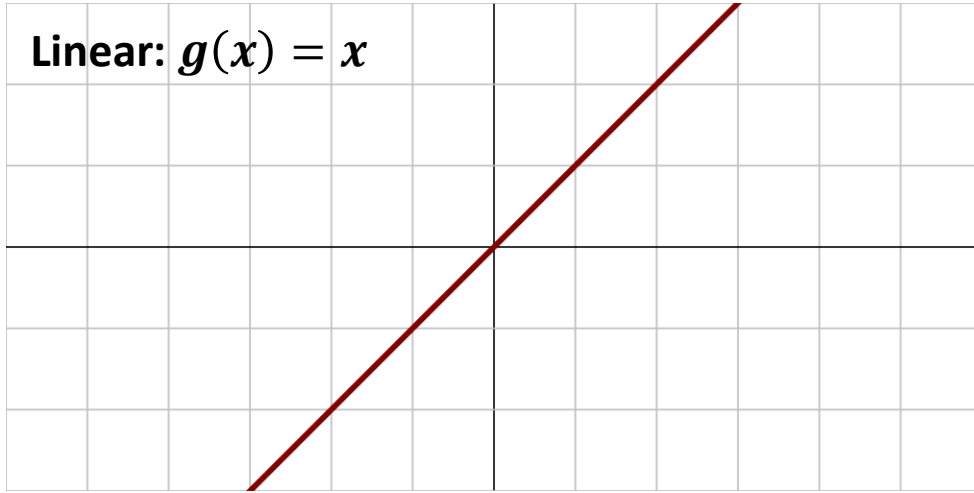
Single layer neural networks

- What about the function $g(\cdot)$?
- This is called the **activation function**
- A transformation of the linear combination of predictors

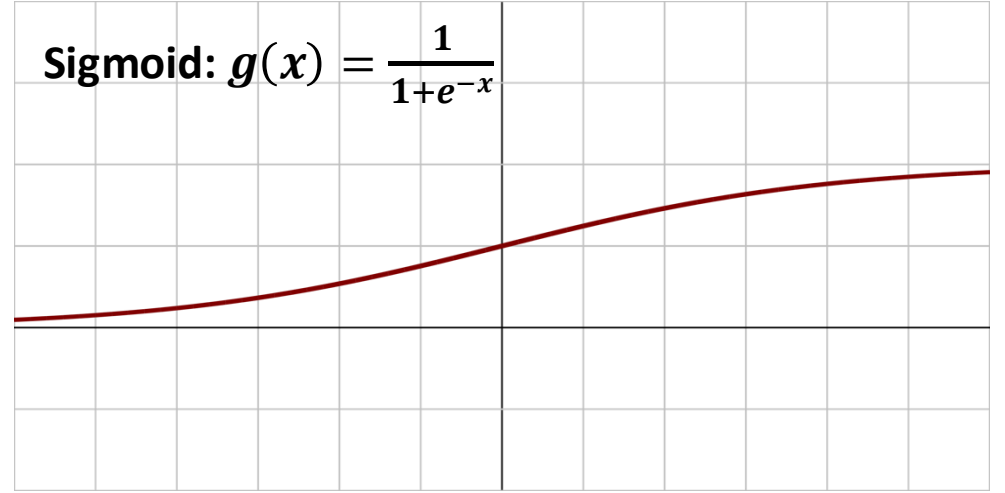
$$h_k(X) = g \left(\beta_k + \sum_{p=1}^P w_{pk} x_p \right)$$

Activation functions

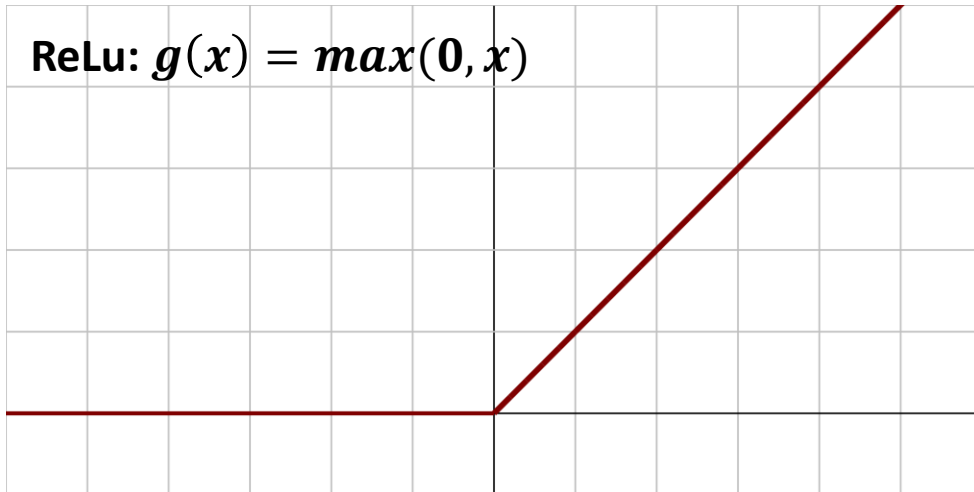
Linear: $g(x) = x$



Sigmoid: $g(x) = \frac{1}{1+e^{-x}}$



ReLU: $g(x) = \max(0, x)$



- Rectified linear (ReLU) is most popular nowadays
- Nonlinearity necessary! Otherwise: just a linear regression

Activation functions

We can go wider

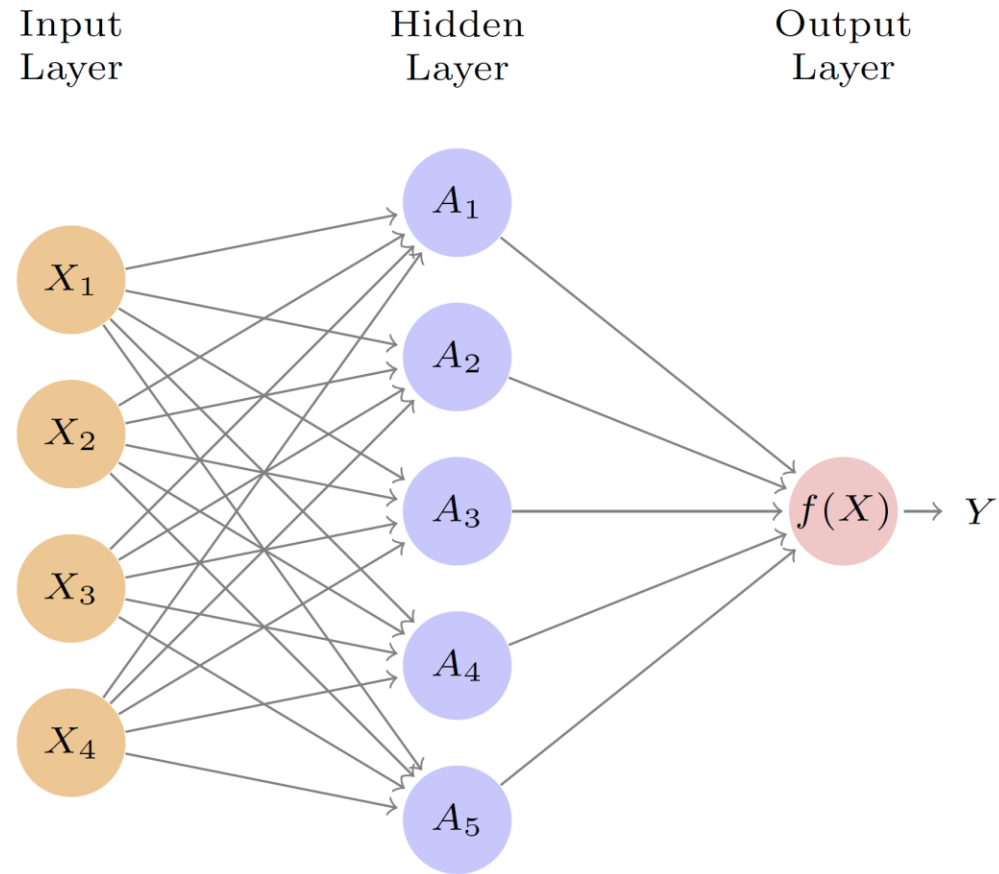
- More hidden units -> more transformations of input

Universal function approximation theorem

Any “well-behaved” function can be represented by neural net of sufficient width with nonlinear activation functions

(you may need an inconvenient number of hidden units!)

Feed-forward neural networks



Feed-forward neural networks

We can go deeper

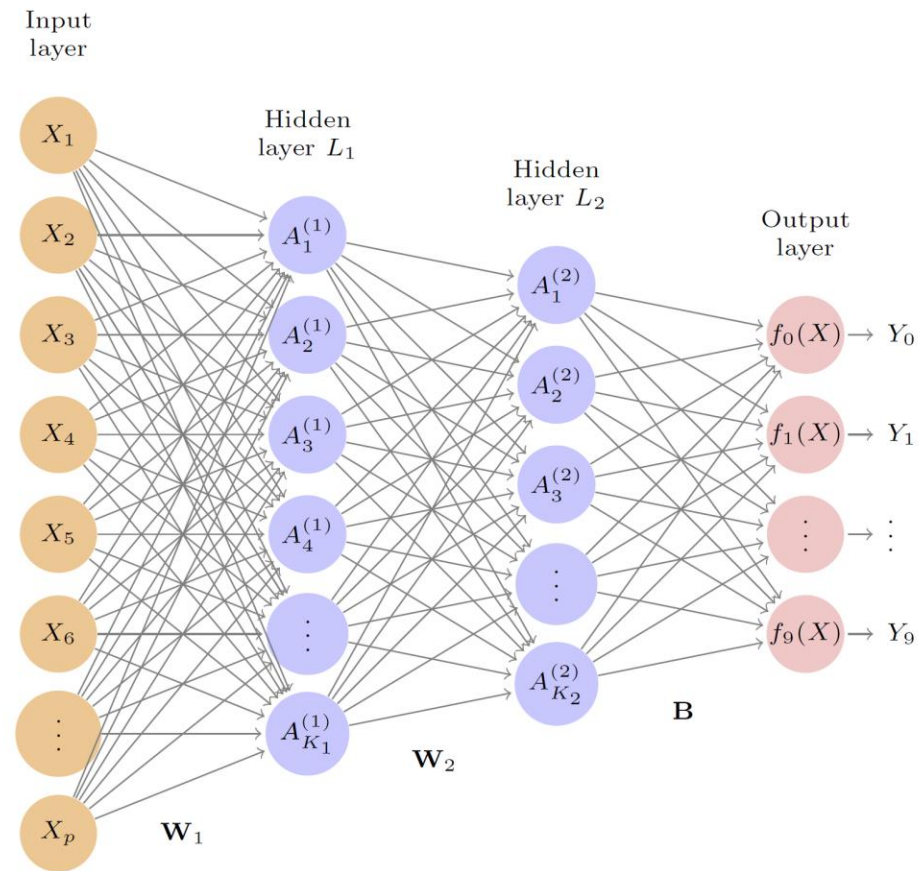
- More hidden layers after one another
- Higher-order features composed of lower-order features

Universal function approximation theorem, version 2

Any “well-behaved” function can be represented by neural net of sufficient *depth* with nonlinear activation functions

(deep neural nets may be more tractable than wide)

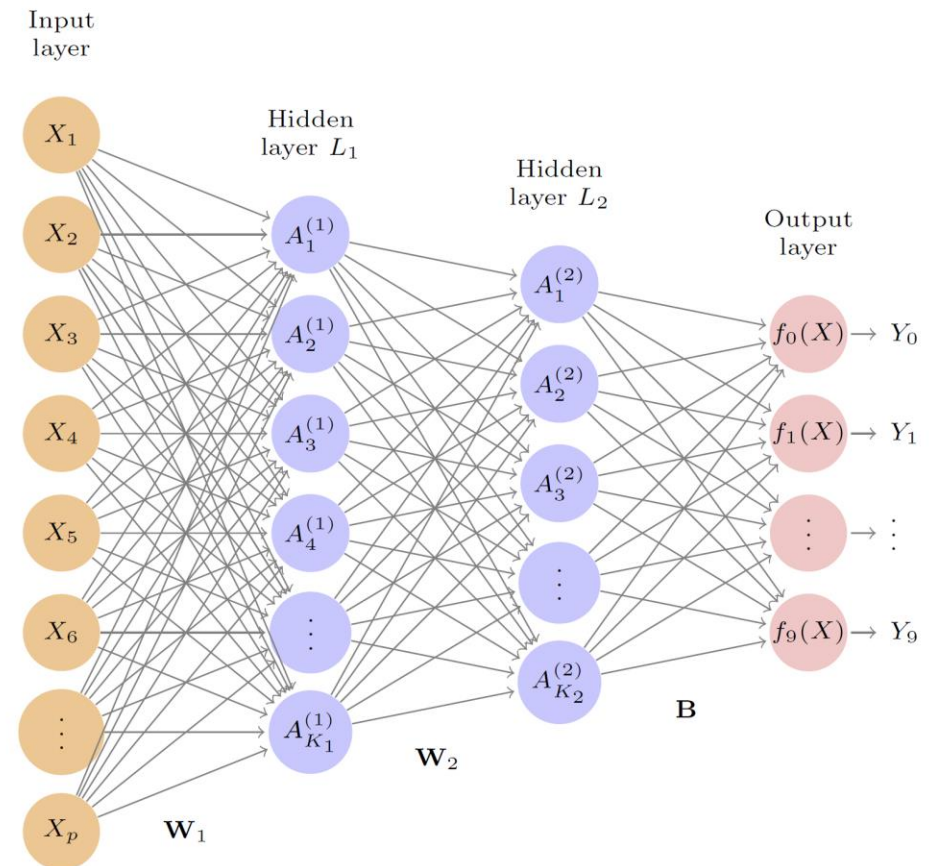
Feed-forward neural networks



Feed-forward neural networks

Feed-forward network **architecture** defined by:

- Number of layers
- Number of hidden units in each layer
- Activation function for each layer
- Activation function for output layer



Keras!

```
library(keras)
```

```
model_dff <-
```

```
  keras_model_sequential() |>
```

```
  layer_flatten(input_shape = c(28, 28)) |>
```

```
  layer_dense(units = 256, activation = "relu") |>
```

```
  layer_dense(units = 128, activation = "relu") |>
```

```
  layer_dense(10, activation = "softmax")
```

Keras!

```
summary(model_dff)
```

Layer (type)	Output Shape	Param #
=====		
flatten (Flatten)	(None, 784)	0

dense_1 (Dense)	(None, 256)	200960

dense_2 (Dense)	(None, 128)	32896

dense_3 (Dense)	(None, 10)	1290
=====		

```
Total params: 235,146
```

```
Trainable params: 235,146
```

```
Non-trainable params: 0
```

How to estimate parameters?

Estimating parameters

- We need some way to measure how well the network does
- Parameters that make the network perform well are good!
- Remember ML estimation: finding $\hat{\theta}$ maximizing $p(y|\hat{\theta})$
- Remember OLS estimation: finding $\hat{\beta}$ minimizing $\sum (y - X\hat{\beta})^2$
- Same for neural nets: we minimize some loss function $L(\theta)$

Loss function

- For continuous outcomes you can use squared error
(same as linear regression!)

$$L(\theta) = (f(X_i; \theta) - y_i)^2$$

- For binary outcomes you can use binary cross-entropy
(same as logistic regression!)

$$L(\theta) = -(y_i \log(f(X_i; \theta)) + (1 - y_i) \log(1 - f(X_i; \theta)))$$

Loss function

- What do the parameters need to be to minimize loss?
- We don't know this!
- But we might know the *direction* in which we need to move to increase the loss
- This direction is called the **gradient** (of loss w.r.t parameters)

$$g(\theta) = \nabla_{\theta} = \frac{\partial}{\partial \theta} L(\theta)$$

- (Looks scary, but it's just a number for each parameter)

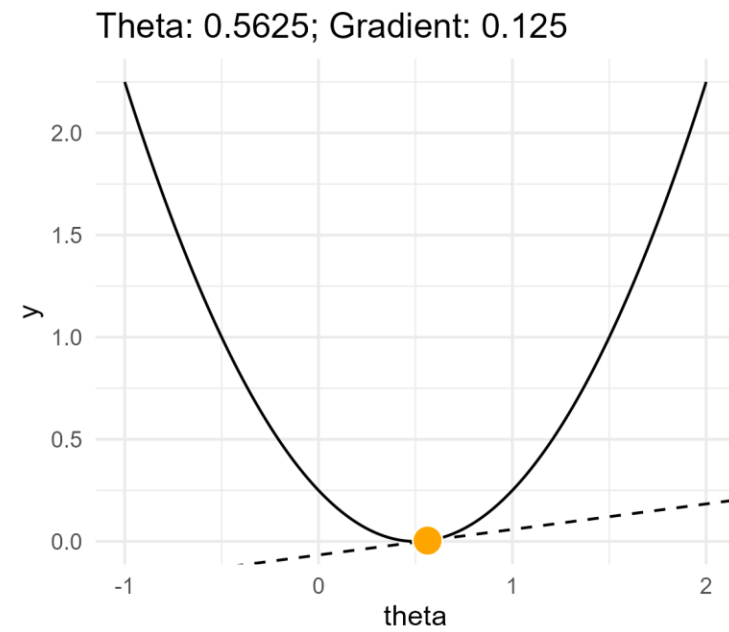
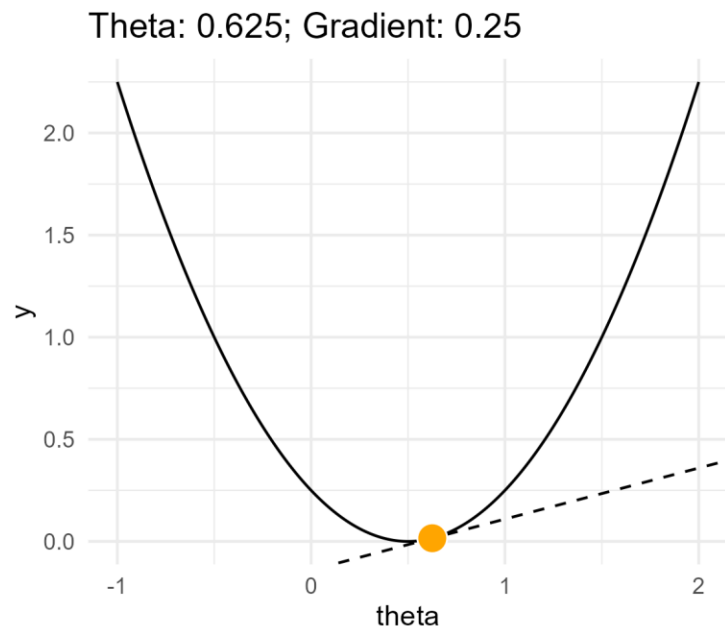
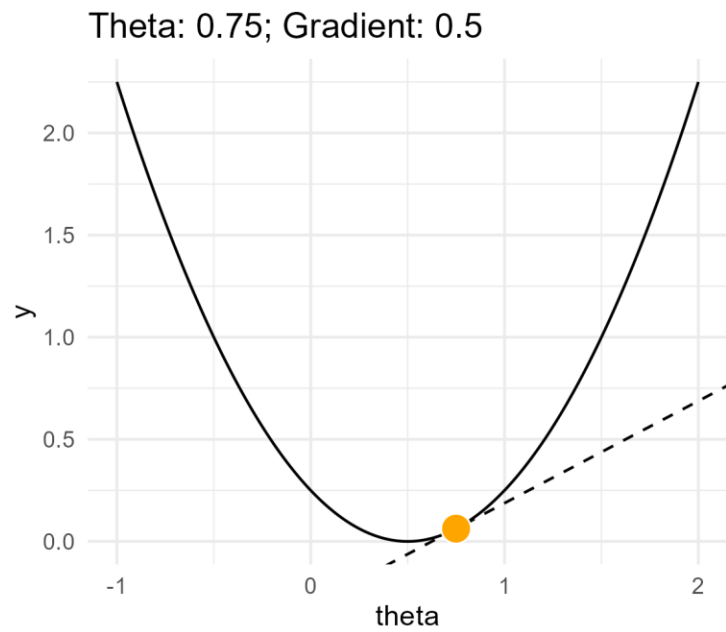
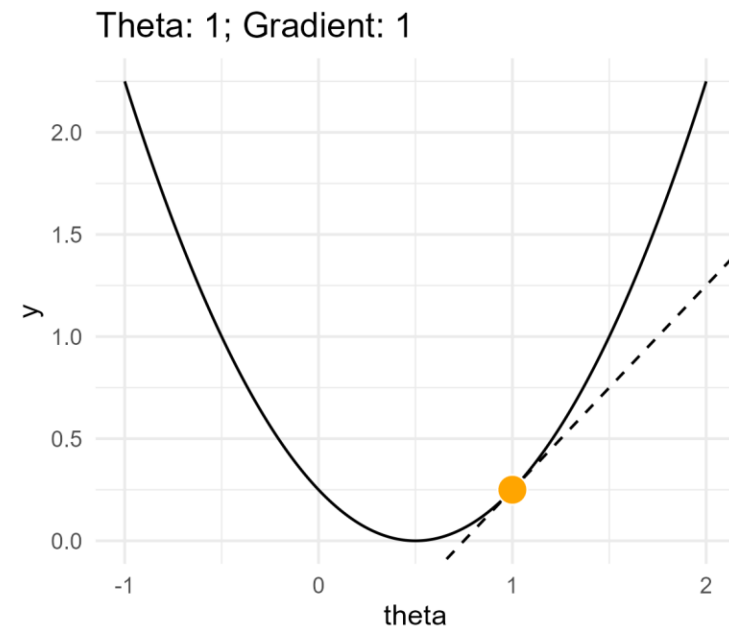
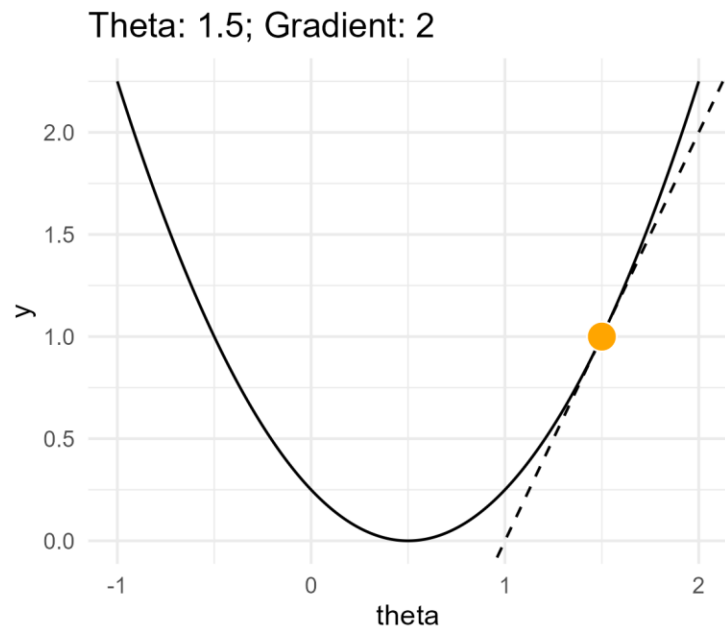
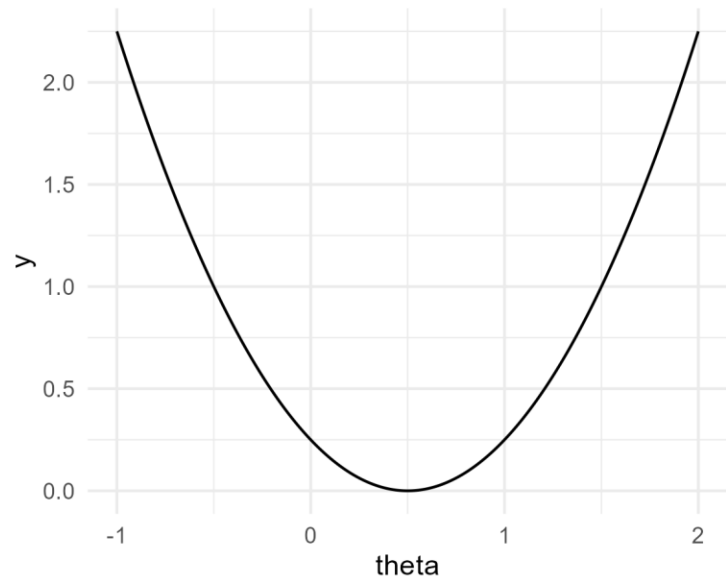
Gradient descent

Iteration: step of size λ in the direction of the negative gradient

$$\theta^{(j+1)} = \theta^{(j)} - \lambda \cdot g(\theta^{(j)})$$

Let's try it out with a simple example!

- $L(\theta) = \theta^2 - \theta + 0.25$
- $g(\theta) = 2\theta - 1$
- $\lambda = 0.25$



Stochastic gradient descent

- Instead of computing the gradients w.r.t. the entire loss function, only use a random **batch** of data
- Take a step after each batch
 - Common batch sizes: 32, 64, 128, 256, 512
- One look at the full data = 1 **epoch**

Stochastic gradient descent

- **(full-)batch mode:** where the batch size is equal to the total dataset thus making the iteration and epoch values equivalent
- **mini-batch mode:** where the batch size is greater than one but less than the total dataset size. Usually, a number that can be divided into the total dataset size.
- **stochastic mode:** where the batch size is equal to one. Therefore, the gradient and the neural network parameters are updated after each sample.

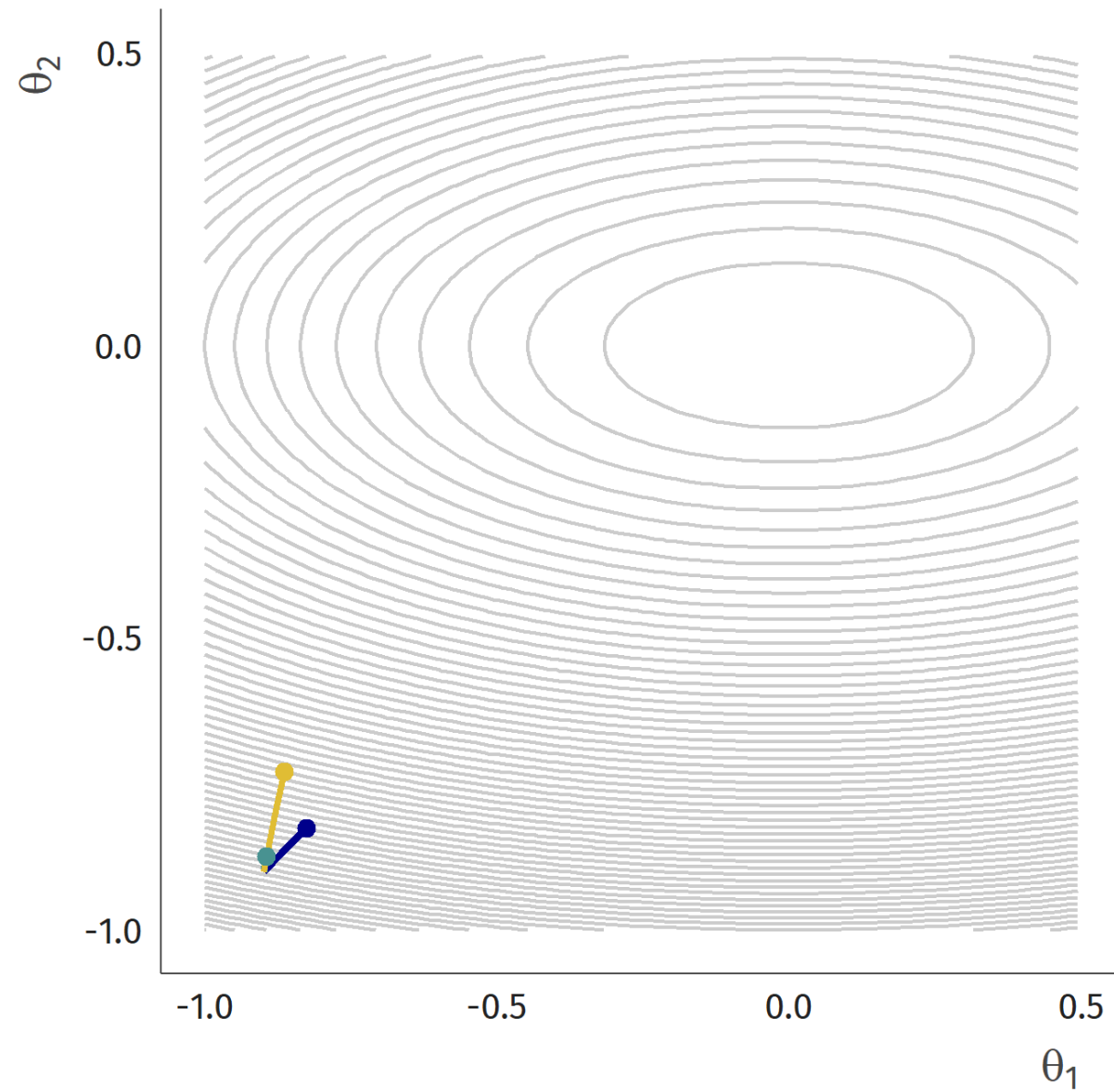
Gradient computation

- But in neural networks, how do we compute gradients?
- We have functions of functions!
- Software like tensorflow / Keras / torch does this for you!
- **Backpropagation**: smart repeated use of the *chain rule* to compute derivatives

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx},$$

- Software also implements gradient descent (and friends)

Adam GD with momentum Gradient descent



Programming pattern: estimation

```
model_dff |>  
  compile(  
    loss = "sparse_categorical_crossentropy",  
    optimizer = "adam"  
  )
```

```
model_dff |>  
  fit(  
    x = X,  
    y = y,  
    batch_size = 32,  
    epochs = 10  
  )
```

Conclusion: estimation

- We need a loss function (e.g., squared error)
- We need gradients (how to change θ to reduce $L(\theta)$)
- Gradient descent: take steps in direction of -gradient
- Stochastic GD: do this with data batches
- Software handles all of this (**black box!!**)
- Advantage: we can focus on the **architecture**

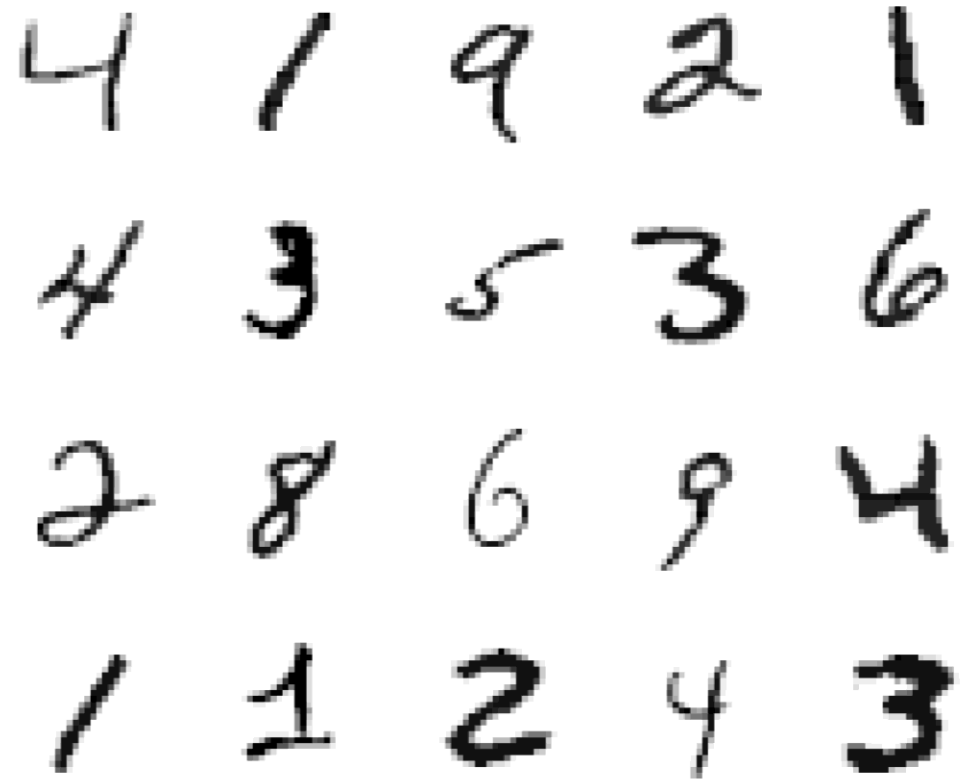
Break

Image processing with convolutional neural networks

Prediction for MNIST

Each example has:

- $28 \times 28 = 784$ input features
 - Values between 0-255 (8 bit)
 - Usually normalized to be 0-1
 - 1 = black, 0 = white, 0.5 = grey
- 10 outcome categories (0-9)
 - One-hot encoding for outcome
 - (cool way to say dummy coding)
 - 1 = 0 1 0 0 0 0 0 0 0 0
 - 5 = 0 0 0 0 0 1 0 0 0 0



What is a convolution

- Convolution is applying a **kernel** (filter) over an image
- The kernel (filter) defines which **feature** is important in the image

What is a convolution

$$\text{Original Image} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix}.$$

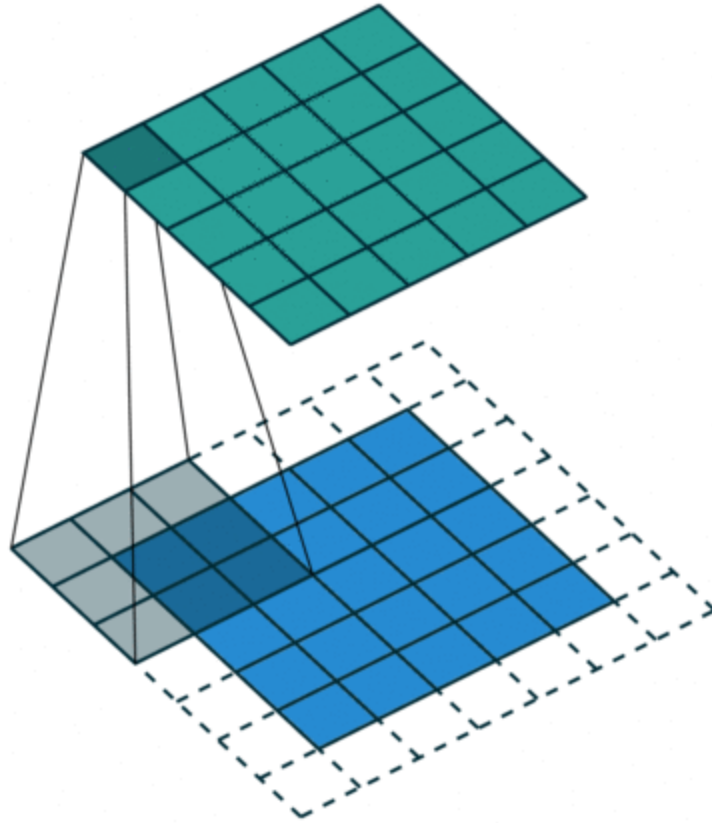
Now consider a 2×2 filter of the form

$$\text{Convolution Filter} = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}.$$

When we *convolve* the image with the filter, we get the result⁸

$$\text{Convolved Image} = \begin{bmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \\ g\alpha + h\beta + j\gamma + k\delta & h\alpha + i\beta + k\gamma + l\delta \end{bmatrix}.$$

What is a convolution



<https://setosa.io/ev/image-kernels/>

Convolution layers

- A convolutional neural network is a NN with one or more **convolution layers**
- The parameters / weights in a convolution layer are the elements of the filter
- The filter is **learnt** by the network!

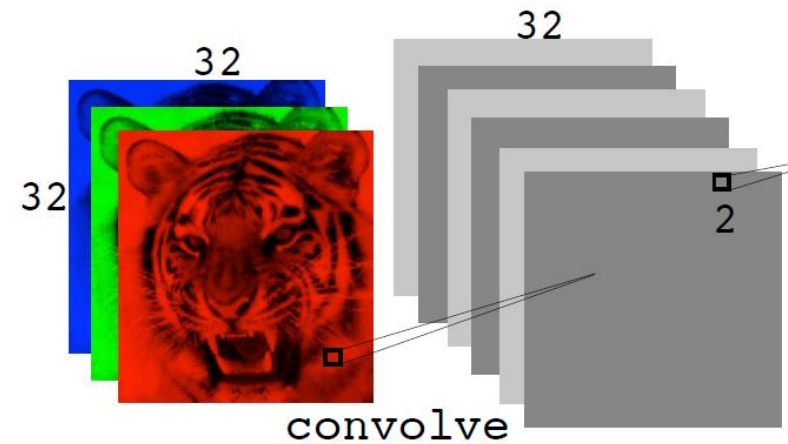


FIGURE 10.8. *Architecture of a convolutional layer. Convolution layers are interleaved with fully connected layers and reduce the size by a factor of 2 in both*

Convolution layers

- In each convolution layer, you can create multiple filters
- Number of parameters is function of:
 - Number of filters (e.g. 6)
 - Size of each filter (e.g. 2x2)
 - NOT the input dimension!
- **Parameter sharing**

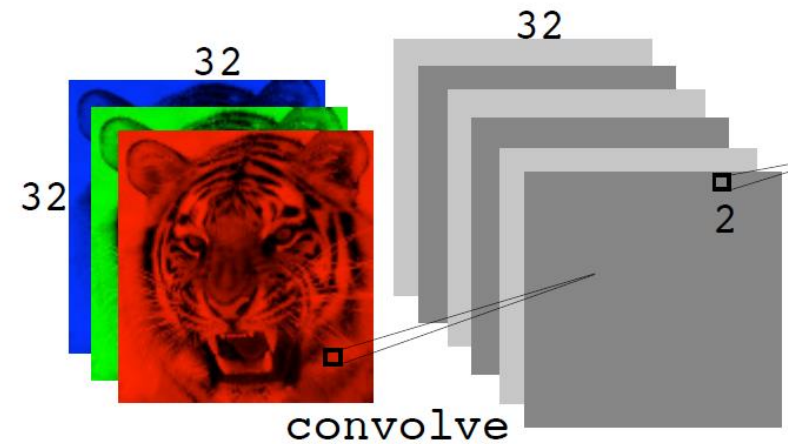


FIGURE 10.8. *Architecture of convolution layers. Convolution layers are integer size by a factor of 2 in both*

Pooling layer

- Convolution layers are usually followed by a **pooling layer**
- Reduces dimensionality
- **Location invariance:**
Robustness against pixel shift / small rotations
- **Max pool** most common

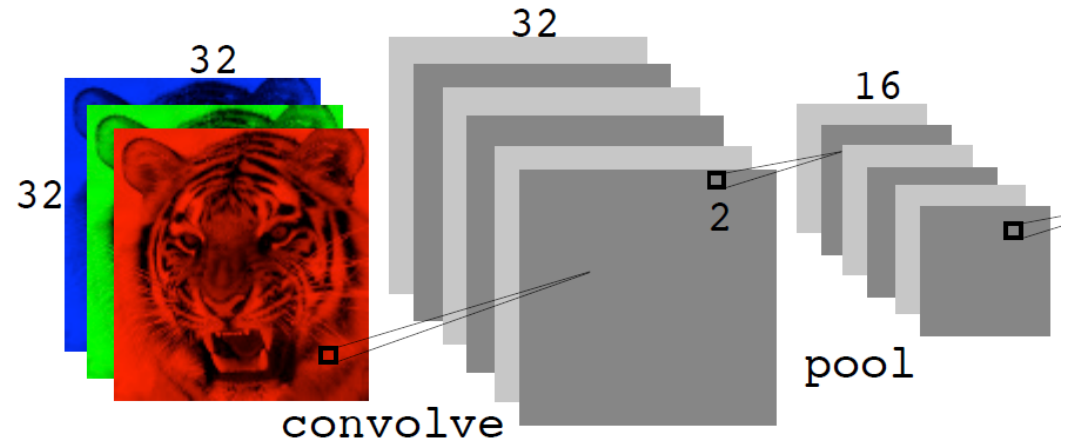


FIGURE 10.8. *Architecture of a deep convolutional neural network. Convolution layers are interspersed with pooling layers. The pooling layer reduces the size by a factor of 2 in both dimensions.*

Pooling layer

$$\text{Max pool} \begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 5 \\ 2 & 4 \end{bmatrix} .$$

Architecture of a CNN

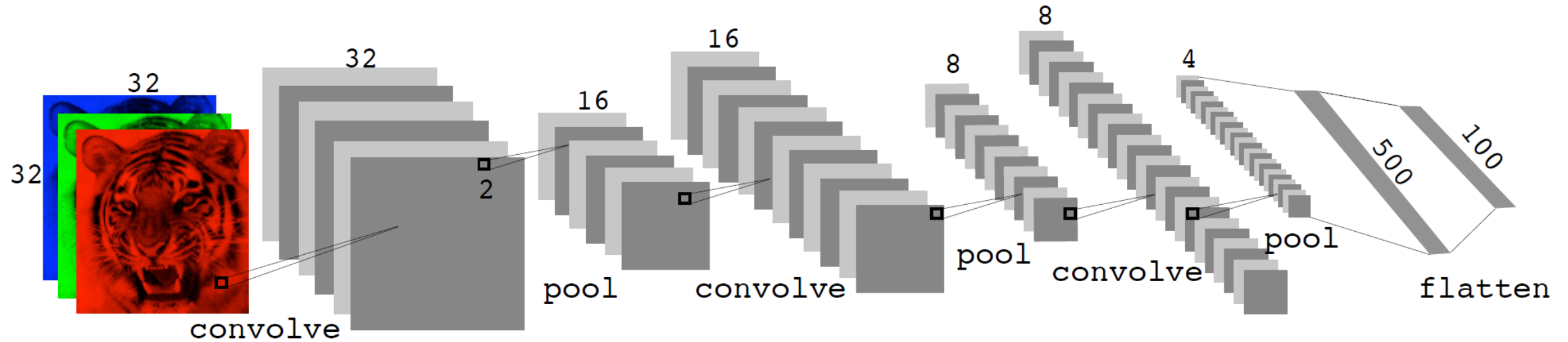


FIGURE 10.8. *Architecture of a deep CNN for the CIFAR100 classification task. Convolution layers are interspersed with 2×2 max-pool layers, which reduce the size by a factor of 2 in both dimensions.*

Applying CNN to MNIST

```
model_cnn <-  
  keras_model_sequential(input_shape = c(28, 28, 1)) |>  
  layer_conv_2d(6, c(5, 5)) |>  
  layer_max_pooling_2d(pool_size = c(4, 4)) |>  
  layer_flatten() |>  
  layer_dense(units = 32, activation = "relu") |>  
  layer_dense(10, activation = "softmax")
```

Model summary

```
summary(model_cnn)
```

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 24, 24, 6)	156
max_pooling2d_2 (MaxPooling2D)	(None, 6, 6, 6)	0
flatten_3 (Flatten)	(None, 216)	0
dense_8 (Dense)	(None, 32)	6944
dense_7 (Dense)	(None, 10)	330
Total params: 7,430		
Trainable params: 7,430		
Non-trainable params: 0		

Compare to DFF model

```
summary(model_dff)
```

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense_1 (Dense)	(None, 256)	200960
dense_2 (Dense)	(None, 128)	32896
dense_3 (Dense)	(None, 10)	1290

```
Total params: 235,146
```

```
Trainable params: 235,146
```

```
Non-trainable params: 0
```

Applying CNN to MNIST

```
model_cnn |>
  compile(
    loss = "sparse_categorical_crossentropy",
    optimizer = "adam"
  )
```

```
model_cnn |>
  fit(
    x = mnist$train$x,
    y = mnist$train$y,
    epochs = 10,
    validation_split = 0.2,
    verbose = 2
  )
```


Performance comparison: DFF

pred										
obs	0	1	2	3	4	5	6	7	8	9
0	975	0	1	0	0	1	0	0	2	1
1	0	1129	1	1	0	1	1	2	0	0
2	4	0	1015	2	0	0	3	2	6	0
3	0	0	6	991	0	4	0	4	1	4
4	3	2	2	0	952	0	5	2	0	16
5	3	0	0	10	0	869	4	0	3	3
6	6	2	0	1	2	4	942	0	1	0
7	2	5	6	2	0	0	0	1004	3	6
8	6	0	2	3	2	4	1	4	946	6
9	4	3	0	2	3	1	1	3	1	991

Performance comparison: CNN

pred										
obs	0	1	2	3	4	5	6	7	8	9
0	971	0	1	0	1	1	2	1	2	1
1	0	1126	2	1	0	0	2	0	4	0
2	1	1	1020	1	1	0	0	1	6	1
3	0	0	2	997	0	5	0	1	2	3
4	0	0	1	0	970	0	0	0	1	10
5	2	0	0	3	0	881	3	0	2	1
6	5	2	0	0	5	2	941	0	3	0
7	1	3	15	3	0	1	0	994	3	8
8	5	0	3	2	0	0	2	2	956	4
9	1	1	0	1	4	5	0	2	6	989

Performance comparison: CNN

```
# accuracy  
sum(diag(cmat_dff)) / sum(cmat_dff)
```

```
#> [1] 0.9814
```

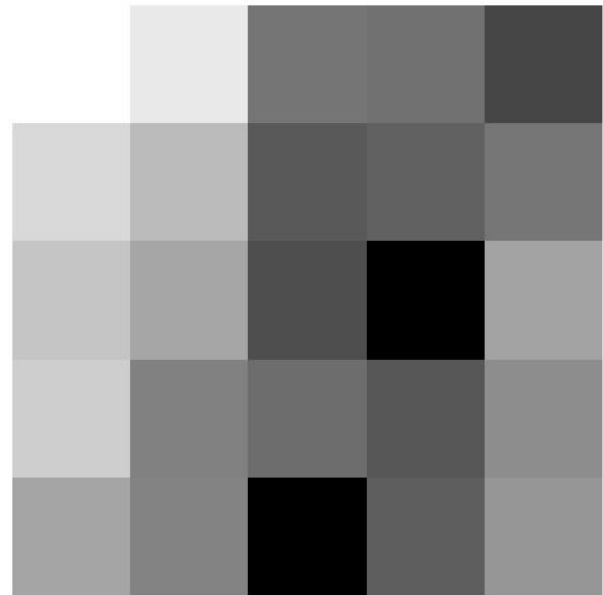
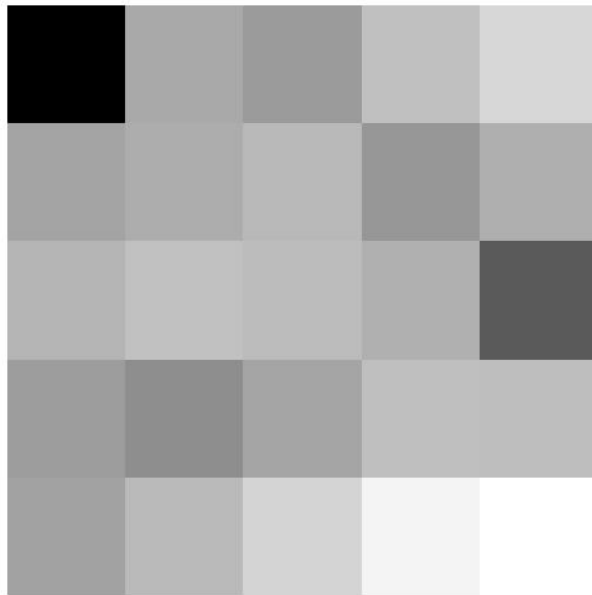
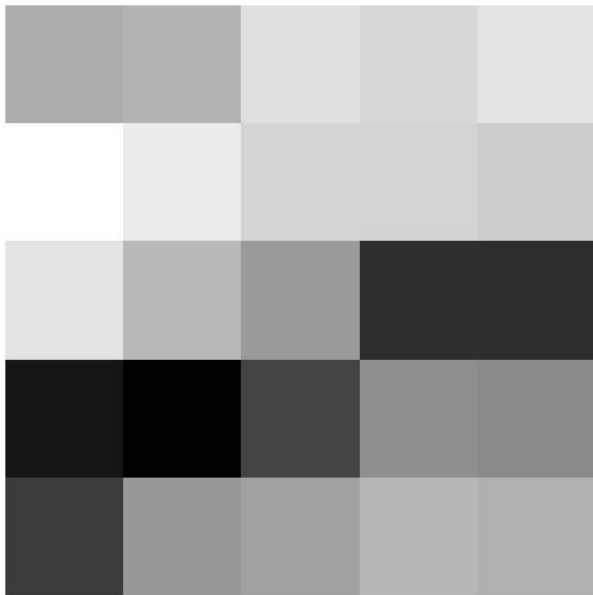
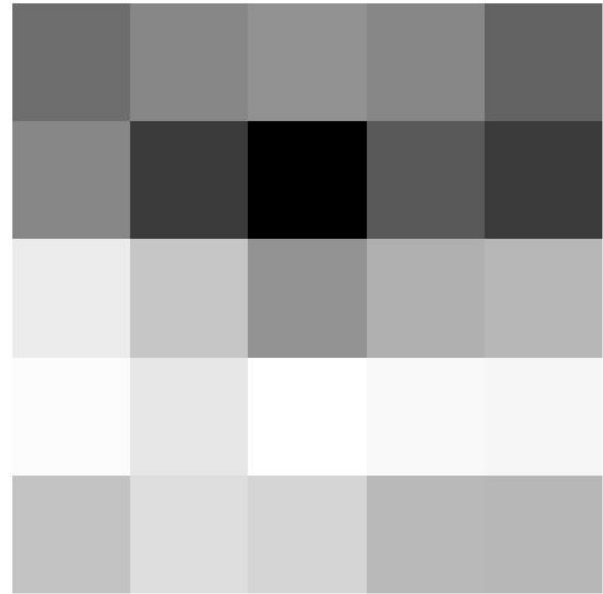
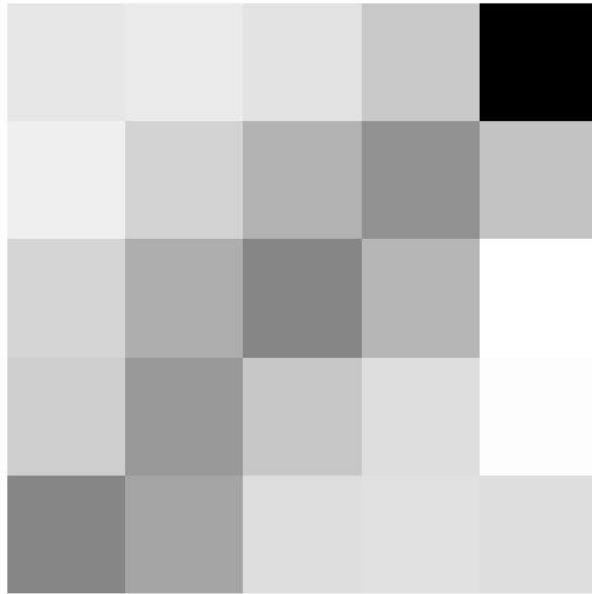
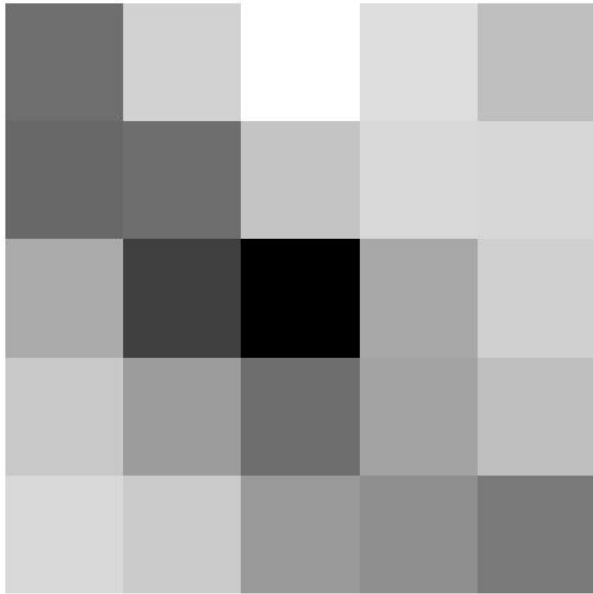
```
sum(diag(cmat_cnn)) / sum(cmat_cnn)
```

```
#> [1] 0.9845
```

What about the features?

Unboxing the black box

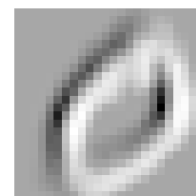
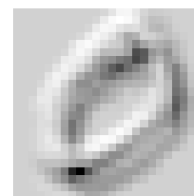
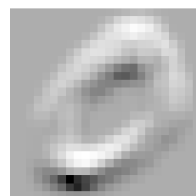
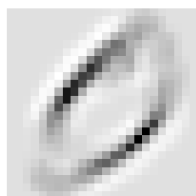
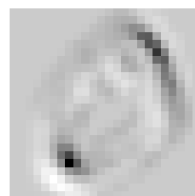
- Extract the weights of the convolution layer to find the features (filters) that were learnt
- Apply the filters to some example images to get an idea of which features are discriminative for the different numbers



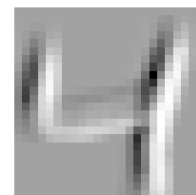
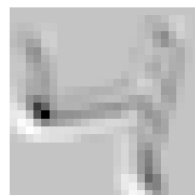
3



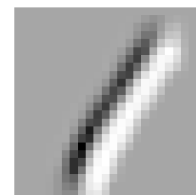
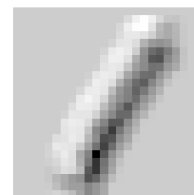
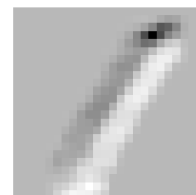
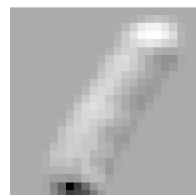
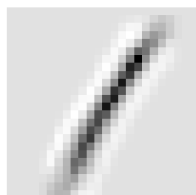
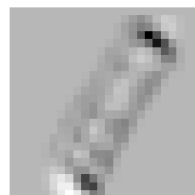
0



5



1



Cool hack: pretrained CNNs

- Download the convolutional layer weights from existing neural network trained on many images
- Apply them to your own images
- Result: a feature vector per image
- Use these feature vectors as input dataset for:
 - Deep feedforward neural network
 - Logistic regression
 - Support vector machine
 - ...
- This can work really well!!

Conclusion: CNN

- Convolution = applying kernels (filters) over an image
- CNNs employ convolution layers
 - Parameter sharing
 - Feature detection
- Followed by pooling layers
 - location invariance
- (Was) state-of-the art in image recognition
- Use pretrained networks as a quick proxy

Battling the curse of dimensionality

Regularization in NNs

- We may have thousands or even millions of parameters
- How can we avoid overfitting?
- How can we fight the curse of dimensionality?
- NNs are not magic: we need **regularization**.
 - Regularization is anything which introduces bias in the parameters to improve generalization (Goodfellow et al., 2016)

Regularization in NNs

- **Convolution:** parameters are set to be equal to one another in different areas of image (parameter sharing)
- **L1 or L2 penalty** applied to weights is common in neural networks (keras can do it!)
- **Dropout regularization:** In each iteration, only update a subset of the parameters
- **Early stopping:** Do not train for many epochs, but only until validation set loss does not improve
- **Data augmentation:** Add shifted / rotated versions of images to input (upside-down tiger is still a tiger!)

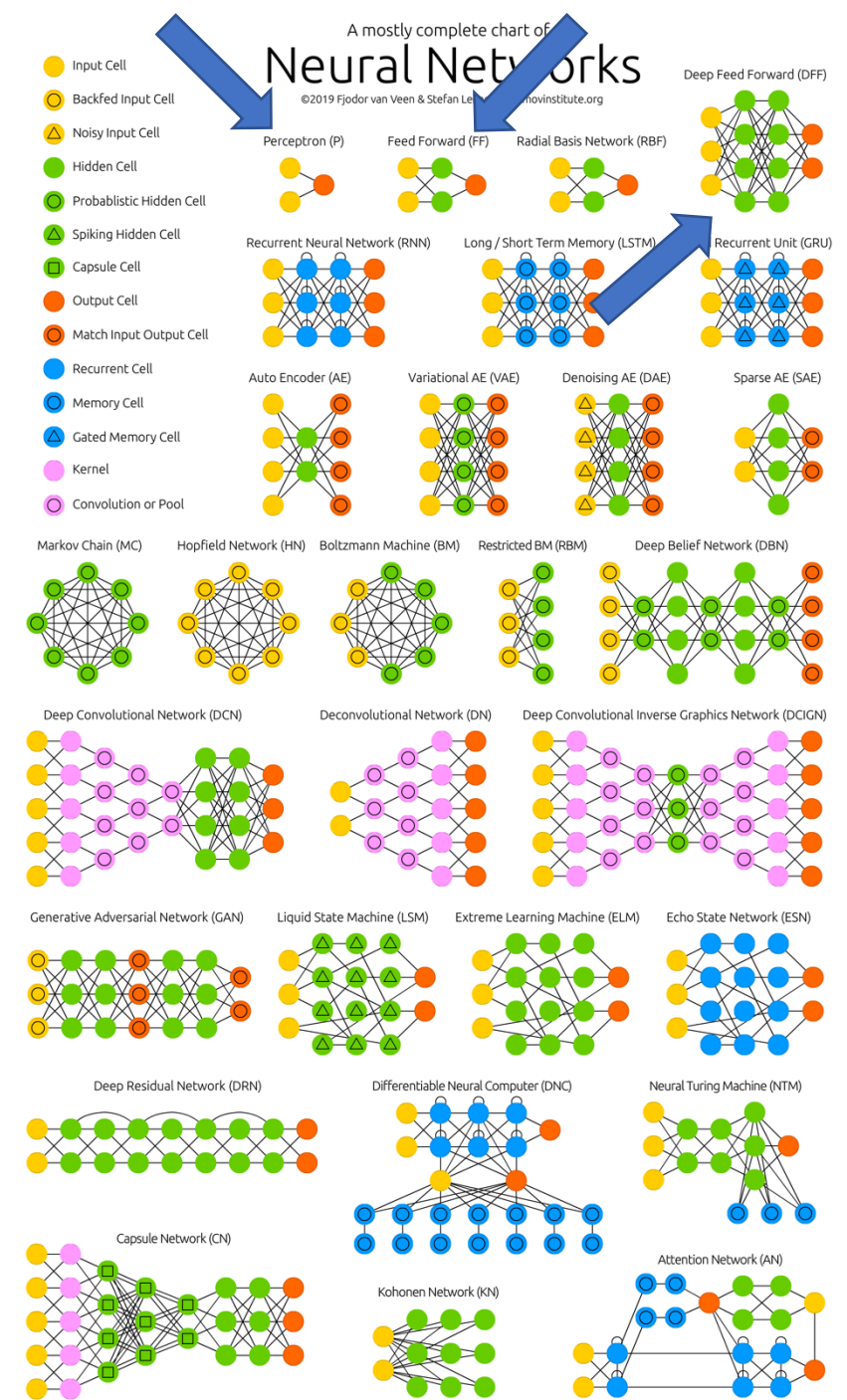
Conclusion

- Introduction to neural networks
- Feed-forward neural networks
- Estimation / optimization
- Convolutional neural networks
- Battling the curse of dimensionality

Epilogue: neural network zoo

Neural network zoo

- You can see how far we got:
 - Perceptron (nonlinear regression)
 - Feed forward
 - Deep feed forward
 - Deep convolutional network
- There is much more 😊



This was just the start

- Recurrent neural networks: for sequences (like text!)
- Autoencoders (nonlinear dimension reduction)
- Generative adversarial networks
<https://thispersondoesnotexist.com/>
- Transformers (engine behind ChatGPT!)
- Look at <https://www.asimovinstitute.org/neural-network-zoo/>

Practical this Friday: image recognition with Keras in R

**Run the take-home part of the
assignment.**