DS 4400

Machine Learning and Data Mining I

Alina Oprea
Associate Professor, CCIS
Northeastern University

November 24 2020

Announcements

- Project milestone: Nov 25
- Ethics of AI: Tue, Dec 1, by Kevin Mills
  - Please complete survey before class
  - Email from Matthew Kopec
- Project presentations
  - Tue, Dec 8, 11:45am-1:25pm
  - Wed, Dec 9, 12:00-2:00pm
- Project report
  - Tue, Dec 15

Outline

- Backpropagation algorithm
  - Example for 2-layer neural network
- Lab on Convolutional Neural Networks
- Regularization
  - Weight decay (aka ridge regularization)
  - Dropout
- Transfer Learning

How to train Neural Networks?

- Backpropagation algorithm
- David Rumelhart, Geoffrey Hinton, Ronald Williams. "Learning representations by back-propagating errors". Nature. 323 (6088): 533–536. 1986
- Applicable to both FFNN and CNN
- Extension of Gradient Descent to multi-layer neural networks

Training Neural Networks

- Training data $x_1, y_1, \dots x_N, y_N$
- One training example $x_i = (x_{i1}, \dots x_{id})$, label y_i
- One forward pass through the network
  - Compute prediction $\widehat{y}_i = h(x_i)$
- Loss function for one example
  - $L(\hat{y}, y) = -[(1-y)\log(1-\hat{y}) + y \log \hat{y}]$

Cross-entropy loss

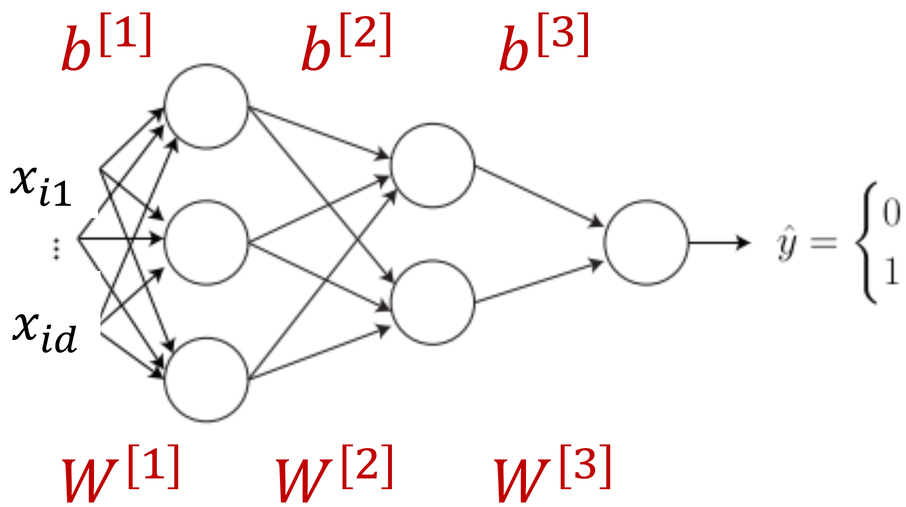
- Loss function for training data
  - $J(W, b) = \frac{1}{N} \sum_i L\,(\widehat{y}_i \;\; , y_i) + \lambda R(W, b)$

GD for Neural Networks

- Initialization
  - For all layers ℓ
    - Initialize $W^{[\ell]}, b^{[\ell]}$
- Backpropagation
  - Fix learning rate α
  - For all layers ℓ (starting backwards)
    - $W^{[\ell]} = W^{[\ell]} - \alpha \sum_{i=1}^{N} \frac{\partial L(\hat{y}_i, y_i)}{\partial W^{[\ell]}}$
    - $b^{[\ell]} = b^{[\ell]} - \alpha \sum_{i=1}^{N} \frac{\partial L(\hat{y}_i, y_i)}{\partial b^{[\ell]}}$

Example 2 Hidden Layers

Training data
Dimension d

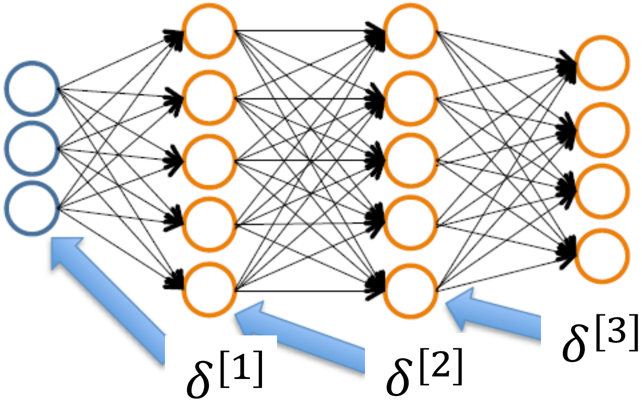


$$z^{[1]} = W^{[1]} x_i + b^{[1]}$$
$$a^{[1]} = g(z^{[1]})$$
$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$
$$a^{[2]} = g(z^{[2]})$$
$$z^{[3]} = W^{[3]} a^{[2]} + b^{[3]}$$
$$\hat{y}^{(i)} = a^{[3]} = g(z^{[3]})$$

Backpropagation

Let $\delta_j^{(l)}$ = "error" of node j in layer l

$$L(y,\hat{y}) = -[(1-y)\log(1-\hat{y}) + y\log\hat{y}]$$



Definitions

- $z^{[\ell]} = W^{[\ell]} a^{[\ell-1]} + b^{[\ell]}, a^{[\ell]} = g\left(z^{[\ell]}\right)$
- $\delta^{[\ell]} = \frac{\partial L(\hat{y},y)}{\partial z^{[\ell]}}$; Output $\hat{y} = a^{[L]} = g\left(z^{[L]}\right)$

1. For last layer L: $\delta^{[L]} = \frac{\partial L(\hat{y},y)}{\partial z^{[L]}} = \frac{\partial L(\hat{y},y)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{[L]}} = \frac{\partial L(\hat{y},y)}{\partial \hat{y}} g'\left(z^{[L]}\right)$
2. For layer ℓ: $\delta^{[\ell]} = \frac{\partial L(\hat{y},y)}{\partial z^{[\ell]}} = \frac{\partial L(\hat{y},y)}{\partial z^{[\ell+1]}} \frac{\partial z^{[\ell+1]}}{\partial a^{[\ell]}} \frac{\partial a^{[\ell]}}{\partial z^{[\ell]}} = \delta^{[\ell+1]} W^{[\ell+1]} g'(z^{[\ell]})$
3. Compute parameter gradients

- $\frac{\partial L(\hat{y},y)}{\partial W^{[\ell]}} = \frac{\partial L(\hat{y},y)}{\partial z^{[\ell]}} \frac{\partial z^{[\ell]}}{\partial W^{[\ell]}} = \delta^{[\ell]} a^{[\ell-1]T}$
- $\frac{\partial L(\hat{y},y)}{\partial b^{[\ell]}} = \frac{\partial L(\hat{y},y)}{\partial z^{[\ell]}} \frac{\partial z^{[\ell]}}{\partial b^{[\ell]}} = \delta^{[\ell]}$

Binary Classification Example

- $\delta^{[3]} = \frac{\partial L(\hat{y},y)}{\partial z^{[3]}} = \frac{\partial L(\hat{y},y)}{\hat{\partial}\ \hat{y}} g'\left(z^{[3]}\right); \hat{y} = g\left(z^{[3]}\right) = a^{[3]}$
- $\frac{\partial L(\hat{y},y)}{\hat{\partial}\ \hat{y}} = - \frac{\partial[(1-y)\log(1-\hat{y})+y log\ \hat{y}]}{\hat{\partial}\ \hat{y}} = \frac{1-y}{1-\hat{y}} - \frac{y}{\hat{y}} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}$
- $\delta^{[3]} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} g'\left(z^{[3]}\right)$

  $= \frac{a^{[3]}-y}{g\left(z^{[3]}\right)\left(1-g\left(z^{[3]}\right)\right)} \mathrm{g}\left(\mathrm{z}^{[3]}\right)\left(1 - \mathrm{g}\left(\mathrm{z}^{[3]}\right)\right) = \mathrm{a}^{[3]} - \mathrm{y}$
- $\frac{\partial L(\hat{y},y)}{\partial\ W^{[3]}} = \delta^{[3]} a^{[2]T} = \left(a^{[3]} - y\ \right) a^{[2]T}$
- $\frac{\partial L(\hat{y},y)}{\partial\ b^{[3]}} = \mathrm{a}^{[3]} - \mathrm{y}$

$$g(x) = \sigma(x) = \frac{1}{1+e^{-x}}$$
$$g'(x) = \sigma'(x) = \sigma(x)(1-\sigma(x))$$

Binary Classification Example

- $\delta^{[2]} = \frac{\partial L(\hat{y},y)}{\partial z^{[2]}} = \delta^{[3]} W^{[3]} g'(z^{[2]})$

- $\frac{\partial L(\hat{y},y)}{\partial\, W^{[2]}} = \delta^{[2]} a^{[1]T} = \delta^{[3]} W^{[3]} g'(z^{[2]})\, a^{[1]T} =$

  $= [a^{[3]} - y] W^{[3]} g(z^{[2]})\, (1 - g(z^{[2]})) a^{[1]T}$

- $\frac{\partial L(\hat{y},y)}{\partial\, b^{[2]}} = [a^{[3]} - y] W^{[3]} g(z^{[2]})\, (1 - g(z^{[2]}))$

$$\boxed{\begin{gathered} g(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \\ g'(x) = \sigma'(x) = \sigma(x)(1 - \sigma(x)) \end{gathered}}$$

Training NN with Backpropagation

Given training set $(x_1, y_1), \ldots, (x_N, y_N)$
Initialize all parameters $W^{[\ell]}, b^{[\ell]}$ randomly, for all layers ℓ
Loop

Set $\Delta_{ij}^{[l]} = 0$, for all layers l and indices i, j

EPOCH

For each training instance (x_k, y_k):

Compute $a^{[1]}, a^{[2]}, \ldots, a^{[L]}$ via forward propagation

Compute errors $\delta^{[L]} = a^{[L]} - y_k, \delta^{[L-1]}, \ldots \delta^{[1]}$

Compute gradients $\Delta_{ij}^{[l]} = \Delta_{ij}^{[l]} + a_j^{[l-1]} \delta_i^{[l]}$

Update weights via gradient step

- $W_{ij}^{[\ell]} = W_{ij}^{[\ell]} - \alpha \frac{\Delta_{ij}^{[\ell]}}{N}$
- Similar for $b_{ij}^{[\ell]}$

Until weights converge or maximum number of epochs is reached

Training Neural Networks

- Randomly initialize weights
- Implement forward propagation to get prediction $\widehat{y_i}$ for any training instance x_i
- Compute loss function $L(\hat{y}_i, y_i)$
- Implement backpropagation to compute partial derivatives $\frac{\partial L(\hat{y}_i, y_i)}{\partial W^{[\ell]}}$ and $\frac{\partial L(\hat{y}_i, y_i)}{\partial b^{[\ell]}}$
- Use gradient descent with backpropagation to compute parameter values that optimize loss
- Can be applied to both feed-forward and convolutional nets

Materials

- Stanford tutorial on training Multi-Layer Neural Networks
  - http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/
- Notes on backpropagation by Andrew Ng
  - http://cs229.stanford.edu/notes-spring2019/backprop.pdf
- Deep learning notes by Andrew Ng
  - http://cs229.stanford.edu/notes2020spring/cs229-notes-deep_learning.pdf

GD for Neural Networks

- Initialization
  - For all layers ℓ
    - Set $W^{[\ell]}, b^{[\ell]}$ at random
- Backpropagation
  - Fix learning rate α
  - For all layers ℓ (starting backwards)
    - $W^{[\ell]} = W^{[\ell]} - \alpha \sum_{i=1}^{N} \frac{\partial L(\hat{y}_i, y_i)}{\partial W^{[\ell]}}$
    - $b^{[\ell]} = b^{[\ell]} - \alpha \sum_{i=1}^{N} \frac{\partial L(\hat{y}_i, y_i)}{\partial b^{[\ell]}}$

This is expensive!

Stochastic Gradient Descent

- Initialization
  - For all layers ℓ
    - Set $W^{[\ell]}, b^{[\ell]}$ at random
- Backpropagation
  - Fix learning rate α
  - For all layers ℓ (starting backwards)
    - For all training examples x_i, y_i

$$-W^{[\ell]} = W^{[\ell]} - \alpha \frac{\partial L(\hat{y}_i, y_i)}{\partial W^{[\ell]}}$$

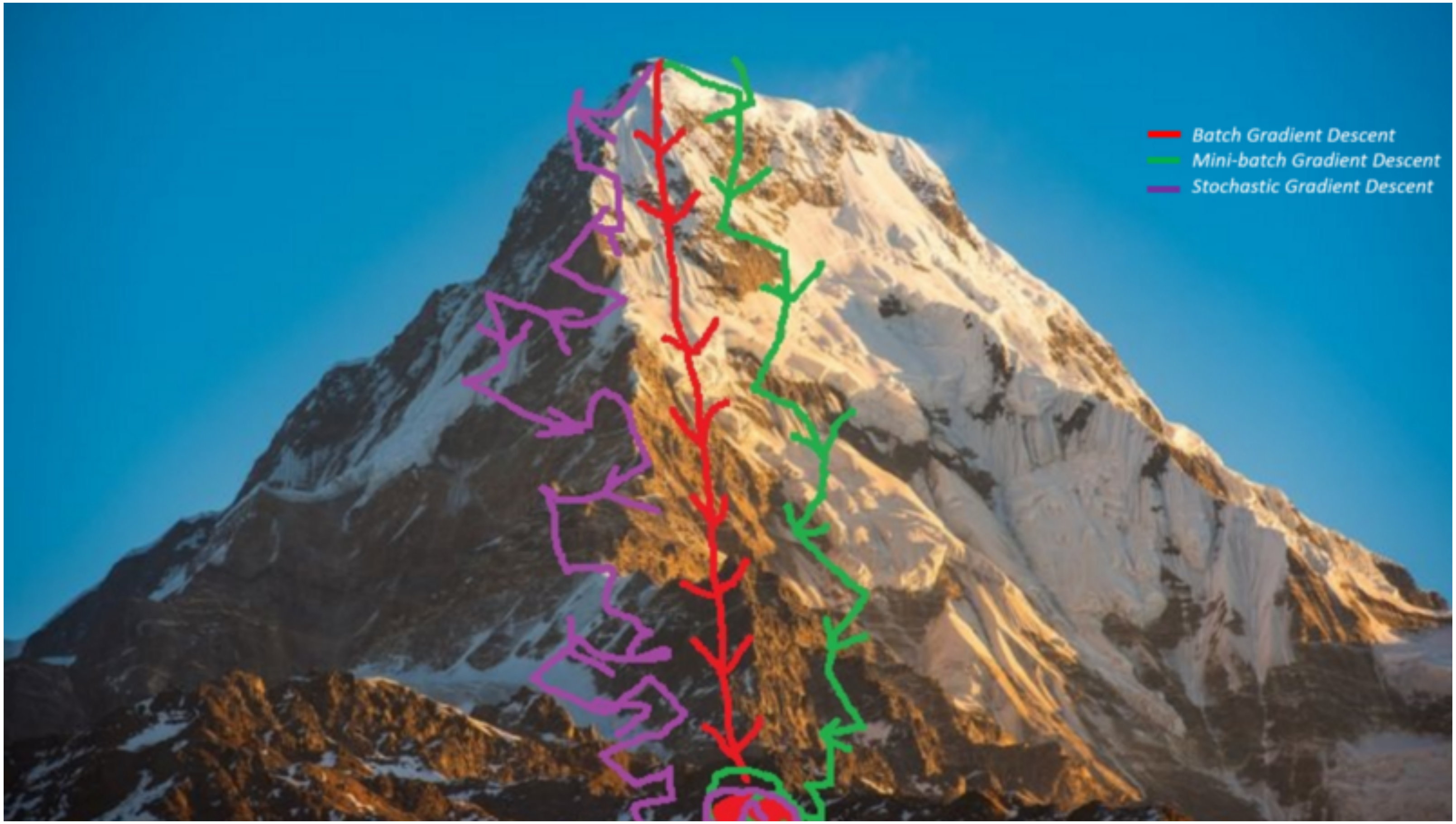
$$-b^{[\ell]} = b^{[\ell]} - \alpha \frac{\partial L(\hat{y}_i y_i)}{\partial b^{[\ell]}}$$

Incremental version of GD

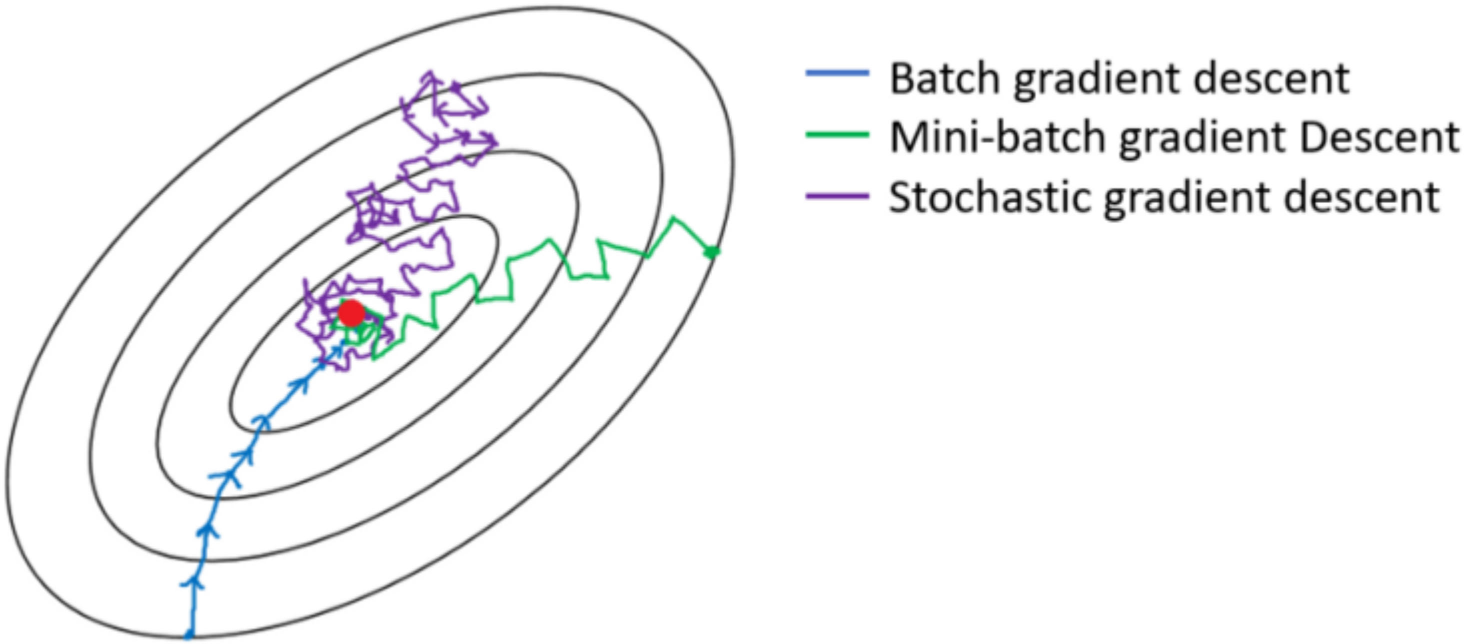
Mini-batch Gradient Descent

- Initialization
  - For all layers ℓ
    - Set $W^{[\ell]}, b^{[\ell]}$ at random
- Backpropagation
  - Fix learning rate α
  - For all layers ℓ (starting backwards)
    - For all batches b of size B with training examples x_{ib}, y_{ib}
      - $W^{[\ell]} = W^{[\ell]} - \alpha \sum_{i=1}^{B} \frac{\partial L(\hat{y}_{ib}, y_{ib})}{\partial W^{[\ell]}}$
      - $b^{[\ell]} = b^{[\ell]} - \alpha \sum_{i=1}^{B} \frac{\partial L(\hat{y}_{ib}, y_{ib})}{\partial b^{[\ell]}}$

Gradient Descent Variants



Gradient Descent Variants



CNN Lab: Load Data

```python
def load_data():
    print("Loading data")
    (X_train, y_train), (X_test, y_test) = mnist.load_data()

    X_train = X_train.astype('float32')
    X_test = X_test.astype('float32')

    X_train /= 255
    X_test /= 255

    y_train = np_utils.to_categorical(y_train, 10)
    y_test = np_utils.to_categorical(y_test, 10)

    X_train = np.reshape(X_train, (60000, 28, 28, 1))
    X_test = np.reshape(X_test, (10000, 28, 28, 1))

    print("Data Loaded")
    return [X_train, X_test, y_train, y_test]
```

Matrix form

Model Architecture

```
def init_model():
    start_time = time.time()

    print("Compiling Model")
    model = Sequential()
    model.add(layers.Conv2D(10, (3, 3), activation='relu', input_shape=(28, 28, 1)))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(5, (3, 3), activation='relu'))
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Flatten())
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(10, activation='softmax'))

    model.summary()

    rms = RMSprop()
    model.compile(loss='categorical_crossentropy', optimizer=rms, metrics=['accuracy'])

    print("Model finished"+format(time.time() - start_time))
    return model
```

10 filters, size 3x3x1

5 filters, size 3x3x10

Vector form

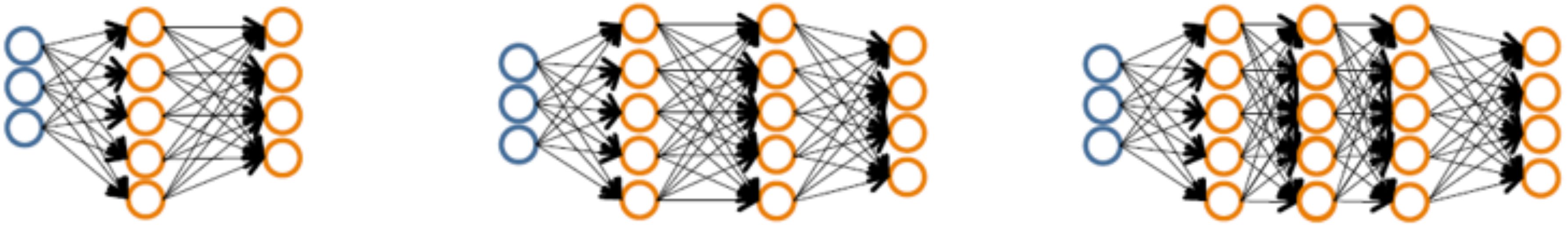
Model Summary

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 10)        100
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 10)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 5)         455
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 5, 5)           0
_________________________________________________________________
flatten_1 (Flatten)          (None, 125)               0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                8064
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
=================================================================
Total params: 9,269
Trainable params: 9,269
Non-trainable params: 0
```

Results

```
totalMemory: 11.90GiB freeMemory: 11.74GiB
2019-03-20 15:23:18.838024: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1308]
2019-03-20 15:23:19.083693: I tensorflow/core/common_runtime/gpu/gpu_device.cc:989]
with 11374 MB memory) -> physical GPU (device: 0, name: TITAN X (Pascal), pci bus id
3s - loss: 0.6465 - acc: 0.8064 - val_loss: 0.3107 - val_acc: 0.9080
Epoch 2/10
1s - loss: 0.2527 - acc: 0.9233 - val_loss: 0.2123 - val_acc: 0.9326
Epoch 3/10
1s - loss: 0.1777 - acc: 0.9466 - val_loss: 0.1556 - val_acc: 0.9550
Epoch 4/10
1s - loss: 0.1386 - acc: 0.9578 - val_loss: 0.1303 - val_acc: 0.9615
Epoch 5/10
1s - loss: 0.1164 - acc: 0.9649 - val_loss: 0.1062 - val_acc: 0.9692
Epoch 6/10
1s - loss: 0.0996 - acc: 0.9697 - val_loss: 0.1032 - val_acc: 0.9677
Epoch 7/10
1s - loss: 0.0882 - acc: 0.9732 - val_loss: 0.0798 - val_acc: 0.9749
Epoch 8/10
1s - loss: 0.0787 - acc: 0.9758 - val_loss: 0.0676 - val_acc: 0.9799
Epoch 9/10
1s - loss: 0.0711 - acc: 0.9783 - val_loss: 0.0680 - val_acc: 0.9804
Epoch 10/10
1s - loss: 0.0664 - acc: 0.9802 - val_loss: 0.0652 - val_acc: 0.9789
Training duration:15.190229892730713
 9760/10000 [============================>.] - ETA: 0s
Network's test loss and accuracy:[0.065167549764638538, 0.9788999999999999]
[alina@dome MNIST]$
```

Overfitting



- The larger the network, the higher the capacity (more model parameters)
- But also more prone to overfitting!

Regularization

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss: Model predictions should match training data}} + \underbrace{\lambda R(W)}_{\text{Regularization: Prevent the model from doing too well on training data}}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

λ = regularization strength (hyperparameter)

<u>L2 regularization:</u> $R(W) = \sum_k \sum_l W_{k,l}^2$ → Weight decay

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

- When computing gradients of loss function, regularization term needs to be taken into account

Dropout

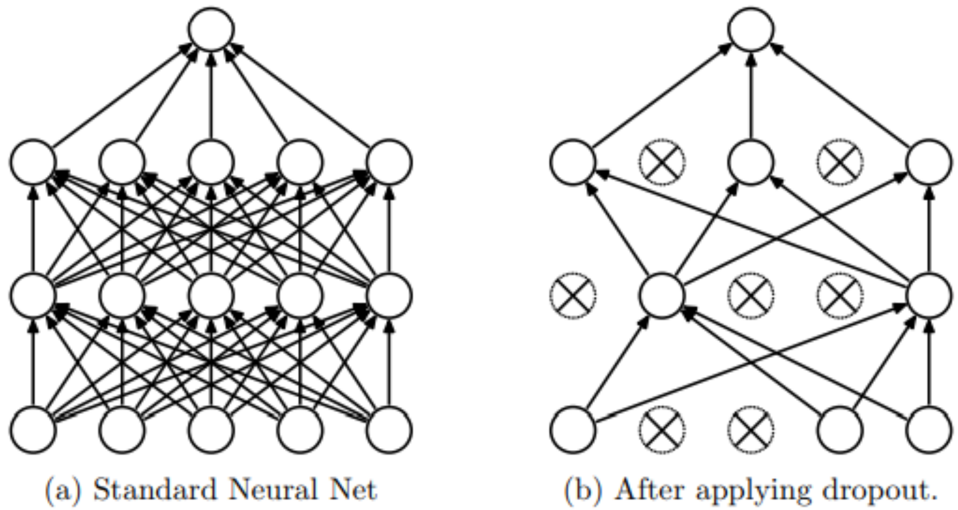
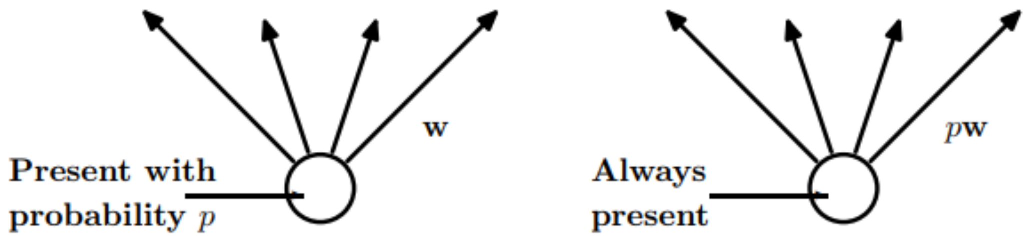


Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

- Regularization technique that has proven very effective for deep learning
- Srivastava et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research 15, 2014

Dropout



(a) At training time (b) At test time

Figure 2: **Left**: A unit at training time that is present with probability p and is connected to units in the next layer with weights $\mathbf{w}$. **Right**: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

- At training time, sample a sub-network and learn weights
  - Keep each neuron with probability p
- At testing time, all neurons are there, but reduce weight by a factor of p

Dropout Implementation

```python
def init_model():
    start_time = time.time()

    print("Compiling Model")
    model = Sequential()

    # Hidden Layer 1
    model.add(Dense(500, input_dim=784))
    model.add(Dropout(0.3))
    model.add(Activation('relu'))

    # Hidden Layer 2
    model.add(Dense(300))
    model.add(Dropout(0.3))
    model.add(Activation('relu'))

    model.add(Dense(10))
    model.add(Activation('softmax'))

    rms = RMSprop()
    model.compile(loss='categorical_crossentropy', optimizer=rms, metrics=['accuracy'])

    print("Model finished"+format(time.time() - start_time))
    return model
```

Dropout regularization

Results on MNIST

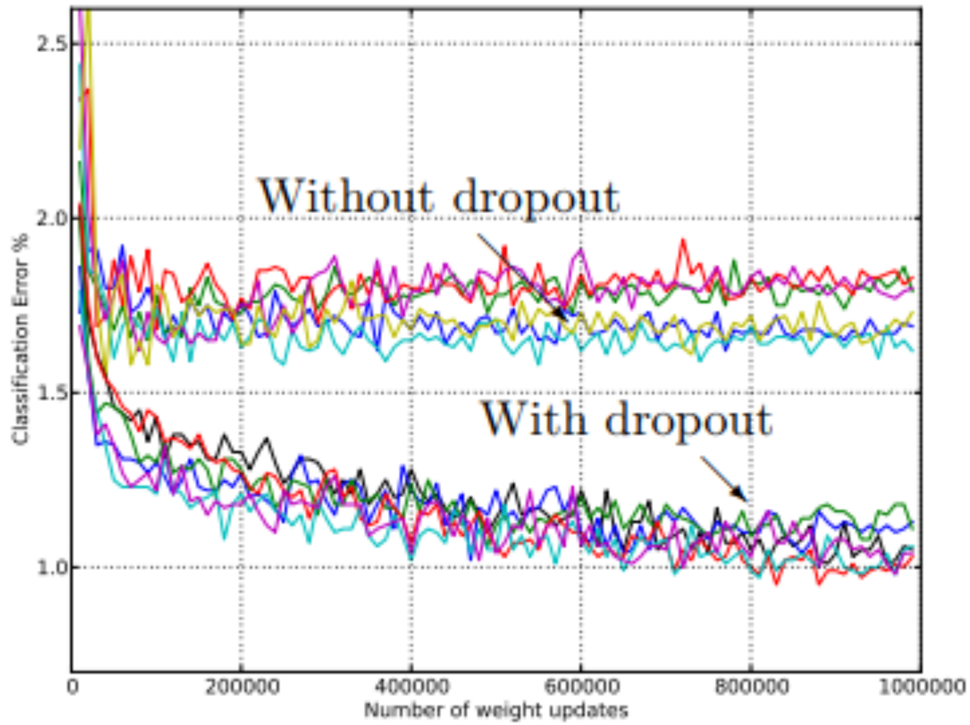


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

Outline

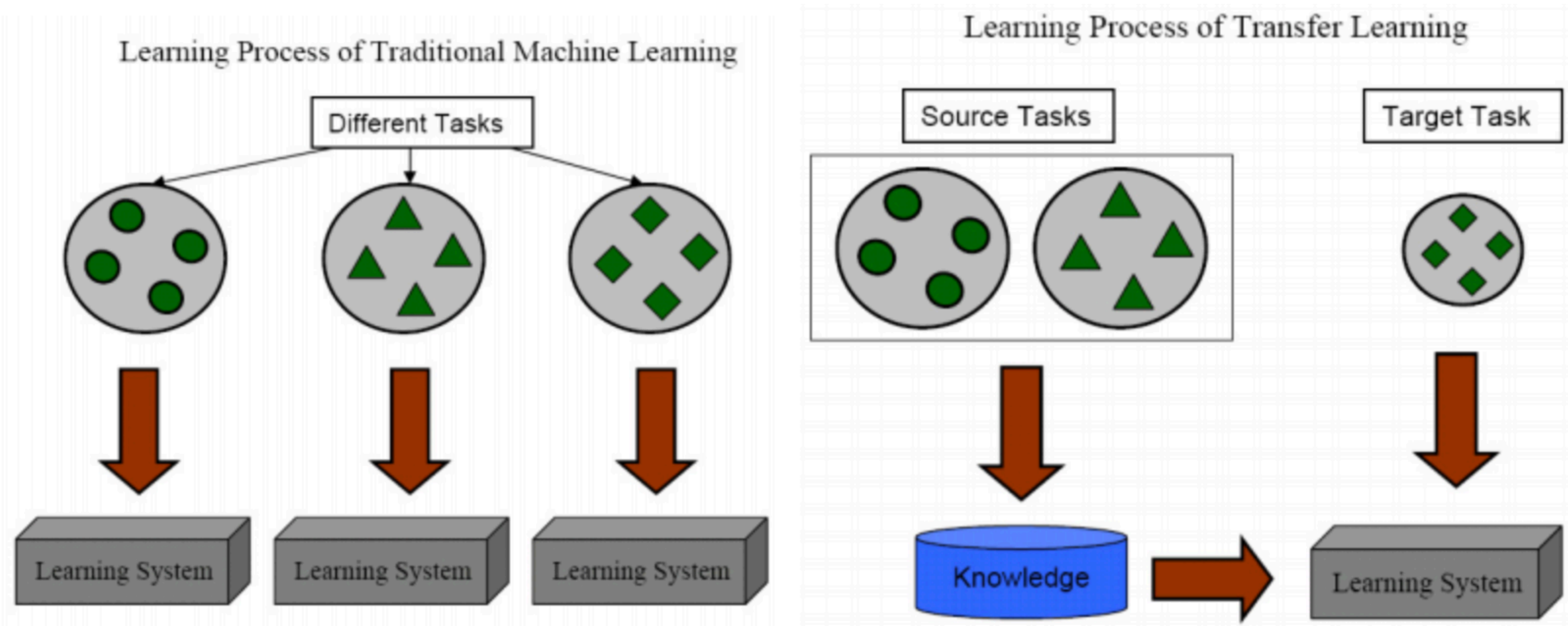
- Backpropagation algorithm
  - Example for 2-layer neural network
- Lab on Convolutional Neural Networks
- Regularization
  - Weight decay (aka ridge regularization)
  - Dropout
- Transfer Learning

Transfer Learning

- Improvement of learning in a **new** task through the *transfer of knowledge* from a **related** task that has already been learned.
- Weight initialization for CNN

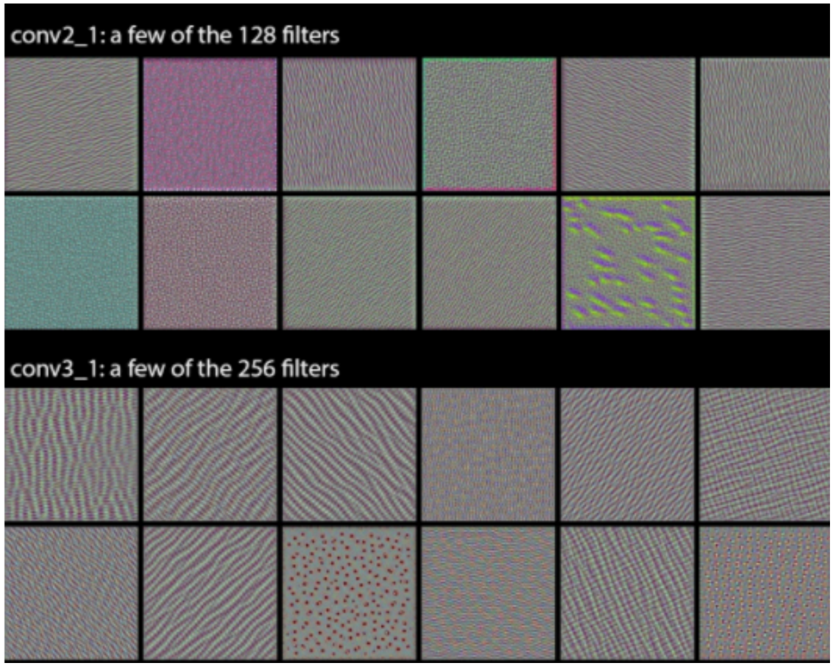
- Two major strategies
  - ConvNet as fixed feature extractor
  - Fine-tuning the ConvNet

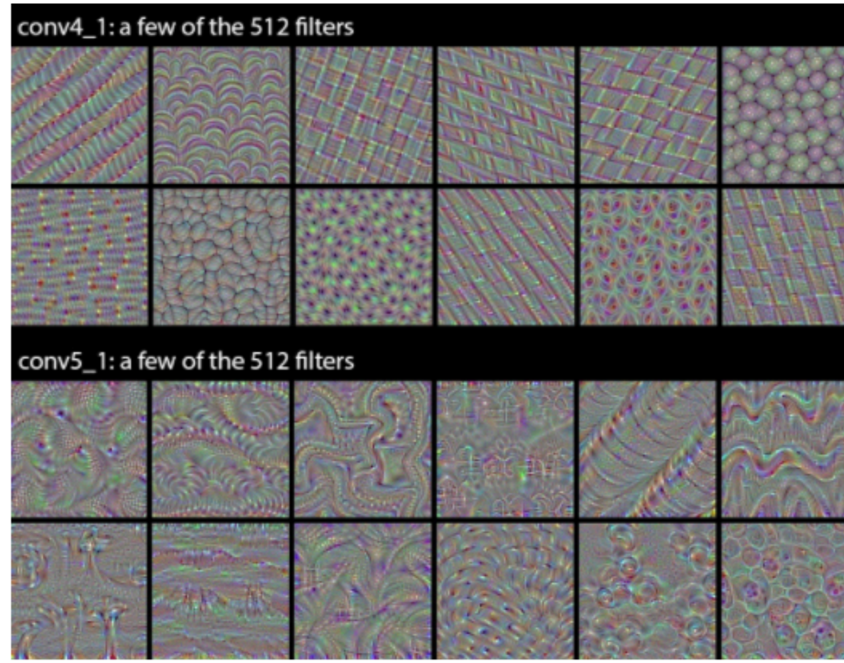
Transfer Learning



(a) Traditional Machine Learning (b) Transfer Learning

Visualizing Filters in VGG 16

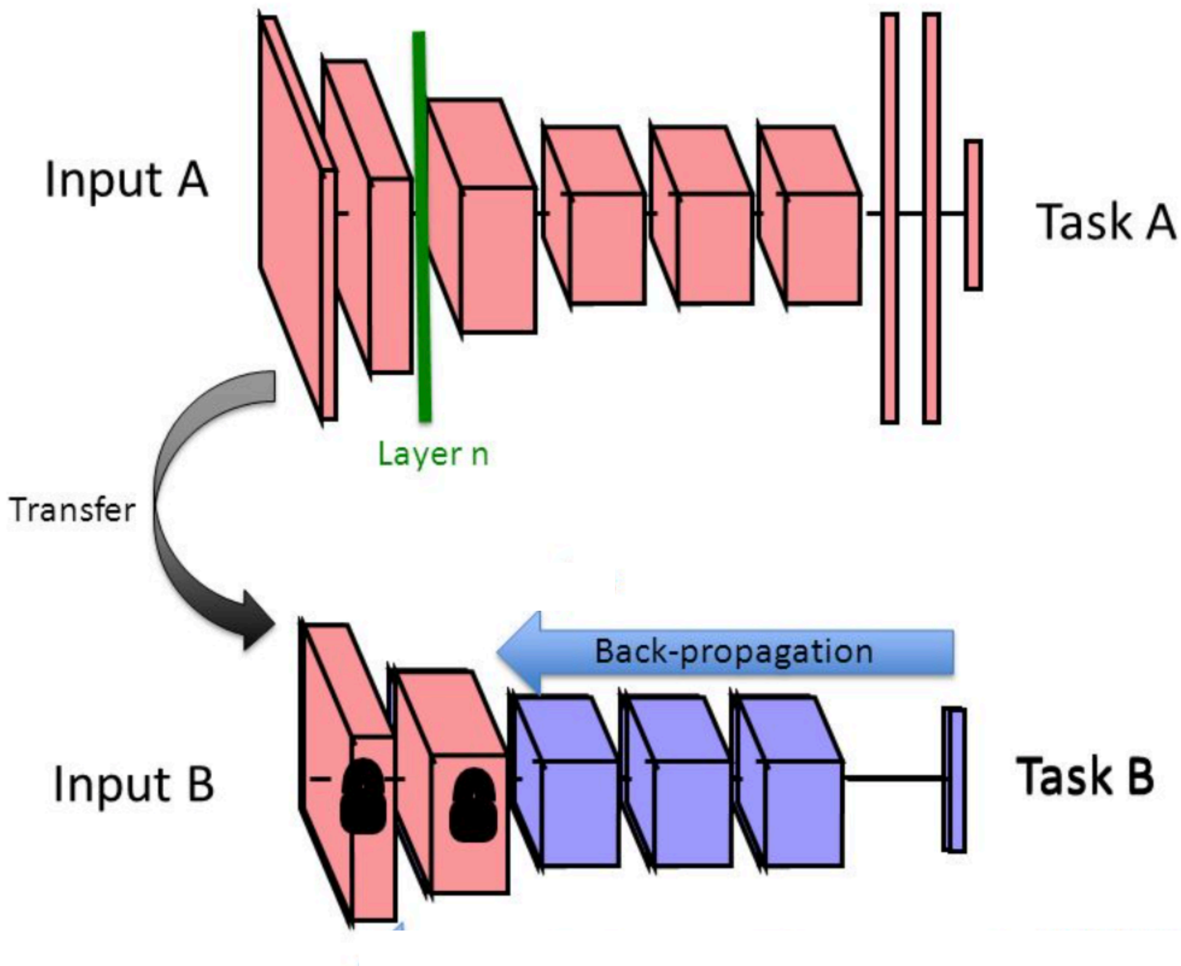




- First layers: general learners
  - Low level notion of edges

- Last layers: specific learners
  - High-level features: eyes, objects

Transfer Learning in NN: Freeze Layers



Methods for Transfer Learning

- Use a pre-trained model
  - https://modelzoo.co/

1. Use Convolutional Nets as Feature Extractor
   - Take a ConvNet pretrained on ImageNet
   - Remove the last fully-connected layer
   - Train the last layer on new dataset (usually a linear classifier such as logistic regression or softmax)
2. Fine-tuning
   - Decide to freeze first n layers
   - Train the remaining layers and stop backpropagation at layer n
   - In the limit fine-tuning can be applied to all layers

How to do Transfer Learning

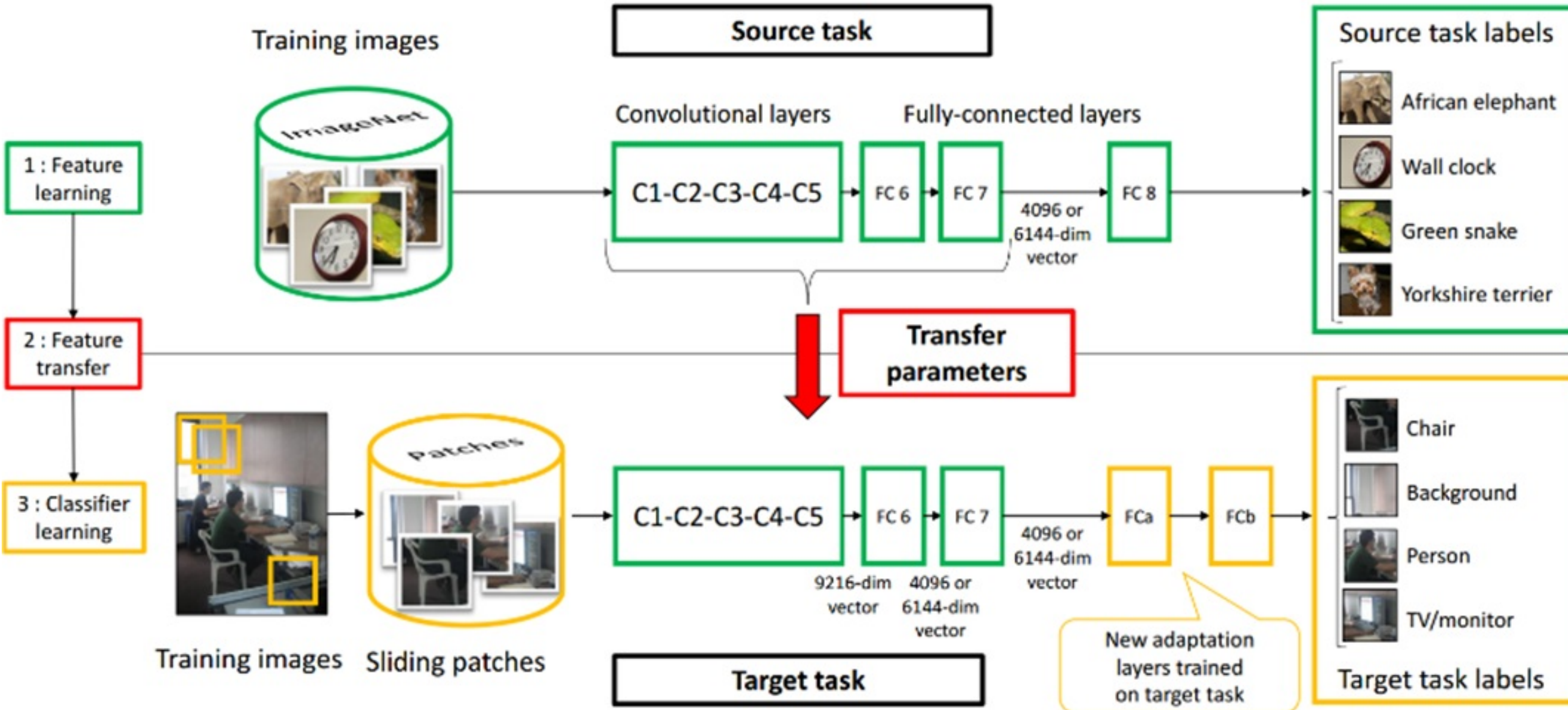
Dataset size	Dataset similarity	Recommendation
Large	Very different	Train model B from scratch, initialize weights from model A
Large	Similar	OK to fine-tune (less likely to overfit)
Small	Very different	Train classifier using the earlier layers (later layers won't help much)
Small	Similar	Don't fine-tune (overfitting). Train a linear classifier

Learning Rates

- Training linear classifier: typical learning rate
- Fine-tuning: use smaller learning rate to avoid distorting the existing weights

Transfer Learning Applications

- Image classification (most common): learn new image classes
- Text sentiment classification
- Text translation to new languages
- Speaker adaptation in speech recognition
- Question answering



Learning and Transferring Mid-Level Image Representations using Convolutional Neural Networks [Oquab et al. CVPR 2014]

Acknowledgements

- Slides made using resources from:
  - Yann LeCun
  - Andrew Ng
  - Eric Eaton
  - David Sontag
  - Andrew Moore
- Thanks!