4-6 Wednesday – 210-GD3

Special topics in Computer Science INT3121 20

Lecturer: Nguyen Thi Ngoc Diep, Ph.D.
Email: ngocdiep@vnu.edu.vn

Slide & Code: https://github.com/chupibk/INT3121-20

Image classification with convolutional neural networks

Week	Content	Class hour	Self-study hour
1 28/8/2019	Introduction Image classification problem and its applications A toy problem with CIFAR10	2	1
2	CNN model architectures and visualization	2	1
3	Training and tuning parameters Automatic parameter learning	2	1
4	Data augmentation Data generator	2	2-6
5	Transfer learning	2	2-6
6	Multi-output image classification	2	2-6
7	Building a training dataset How to write a report	1	2-6
8, 9, 10, 11	Seminar: Bag of tricks with CNN (as mid-term tests)	1	2-6
12, 13, 14	Final project presentations	1-3	2-6
15	Class summarization	1	open

Week 2 recall

- Convolution = combining two functions
- Types of convolution kernels/filters:
  - (Traditional) Convolution
    - Too many parameters
  - Dilated or Atrous convolution
    - Large receptive field
    - Less parameters
  - Separable convolution
    - Even lesser parameters
  - Deconvolution or transposed convolution
    - Low resolution -> high resolution

  - Pooling -> not a convolution layer, but...

Week 2: Homework checklist

- Code Inception and ResNet models
- Confirm code of other models (LeNet, AlexNet, VGG16)
- Run for the CIFAR10 dataset

→ Done? ☺

Steps in training a CNN

- Prepare data: x_train, y_train, x_val, y_val, x_test, y_test
  - x: (b, h, w, c)
  - y: (b, n)
- Define a CNN model
  - Conv + dense layers
- Compile the model
  - Define a loss function: cross entropy loss
  - Set an optimizer & its parameters
  - Define evaluation metrics

model.compile(...)

- Train the model
  - Feed the actual data and do weight updating
    - Feed several times (# of epochs)
    - Feed a small amount at a time (batch_size)

model.fit(...)

What "model.compile()" does?

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])
```

Loss function: Cross entropy

p(x): true distribution
q(x): estimated distribution
x: discrete variable

$$H(p, q) = -\sum_{\forall x} p(x) \log(q(x))$$

Other form:

y: ground truth vector
$\hat{y}$: estimated vector
$(\cdot)$: vector dot product

$$L = -\mathbf{y} \cdot \log(\hat{\mathbf{y}})$$

Loss function example

```
def cross_entropy(y_true, y_pred, eps=1e-12):
    '''
    Log loss is undefined for p=0 or p=1,
        so probabilities are
        clipped to (eps, 1 - eps).
    '''
    y_true = np.clip(y_true, eps, 1. - eps)
    y_pred = np.clip(y_pred, eps, 1. - eps)
    log_likelihood = -np.log(y_pred)
    loss = -(y_true * np.log(y_pred)).sum()
    return loss
```

```
cross_entropy([1, 0, 0, 0], [0.9, 0, 0, 0.1])
```

0.10536051571528555

```
cross_entropy([1, 0, 0, 0], [0.1, 0.8, 0.1, 0])
```

2.3025850930218996

Loss function for a dataset of size N

$$J = -\frac{1}{N}\left(\sum_{i=1}^{N} \mathbf{y_i} \cdot \log(\hat{\mathbf{y}}_\mathbf{i})\right)$$

Metrics

total N=150	Predicted YES	Predicted NO
Actual: YES	True positive TP = 80	False Negative FN = 25
Actual: NO	False Positive FP = 10	True negative TN = 35

$$precision = \frac{number\ of\ correct\ positive\ predictions}{number\ of\ positive\ predictions\ made}$$

$$= \frac{TP}{TP+FP} = \frac{80}{80+10} = 0.89$$

$$accuracy = \frac{number\ of\ correct\ predictions}{total\ number\ of\ predictions\ made}$$

$$= \frac{TP+TN}{N} = \frac{80+35}{150} = 0.7$$

$$recall = \frac{number\ of\ correct\ positive\ predictions}{number\ of\ all\ positive\ samples}$$

$$= \frac{TP}{TP+FN} = \frac{80}{80+25} = 0.76$$

$$F1 = 2 * \frac{1}{\frac{1}{precision} + \frac{1}{recall}}$$

Harmonic mean between precision and recall

What is an optimizer?

Prediction

$$\hat{y} = f(x; w, b) = f(x; \theta) \text{ where } \theta = [w, b]$$

w: weights
b: biases

Loss function

$$L(\theta) = -y \cdot log(\hat{y})$$

Objective:

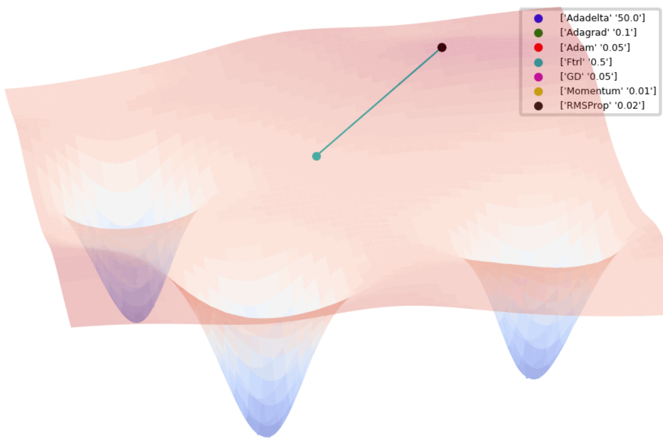
$$min_{\theta} L(\theta)$$

How?

Start from a random position of theta
Iteratively update it

$$\theta = \theta + \eta \Delta\theta$$

Iterative weight updating

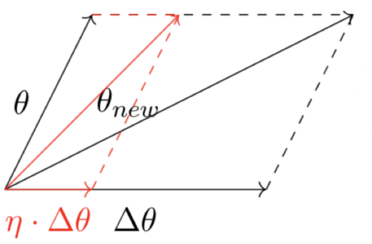


Visualization credit: https://github.com/Jaewan-Yun/optimizer-visualization

Optimizers

- Gradient descent (GD)
- Stochastic gradient descent (SGD)
- Mini-batch gradient descent
- Momentum gradient descent
- Adaptive gradient descent (AdaGrad)
- Root mean square propagation gradient descent (RMSProp)
- Adaptive moment estimation (Adam)

Gradient descent



$\theta new = \theta + \eta.\Delta\theta$

$$L(\theta+\eta\Delta\theta) = L(\theta) + \eta * \underbrace{\overbrace{\boxed{\Delta\theta}}^{=\,u}{}^T \overbrace{\boxed{\nabla L(\theta)}}^{\text{gradient}}}_{<0} + \underbrace{\frac{\eta^2}{2!} * \Delta\theta^T \nabla^2 L(\theta)}_{\to\ 0} + \ldots$$

$$-1 \leq cos(\beta) = \frac{u^T \nabla_\theta \mathscr{L}(\theta)}{||u|| * ||\nabla_\theta \mathscr{L}(\theta)||} \leq 1$$

minimum when going the opposite direction to the gradient

Gradient descent update rule

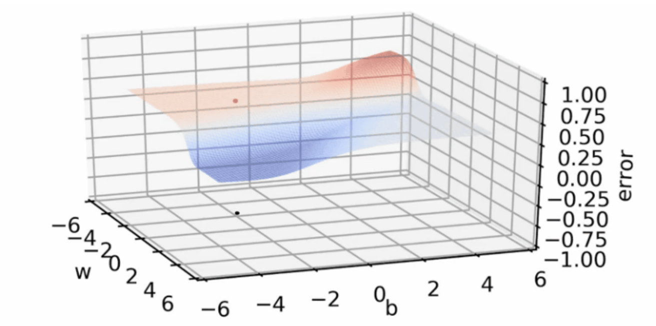
- Randomly initialize w,b
- Iterate over data
  - Compute y-hat
  - Compute loss L(w,b)
  - Update

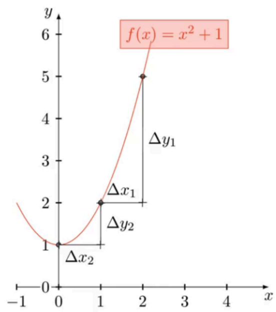
$$w_{t+1} = w_t - \eta \nabla w_t$$
$$b_{t+1} = b_t - \eta \nabla b_t$$
$$where, \nabla w_t = \frac{\partial \mathscr{L}(w,b)}{\partial w}_{at\ w = w_t, b = b_t}, \nabla b = \frac{\partial \mathscr{L}(w,b)}{\partial b}_{at\ w = w_t, b = b_t}$$

Number of updates in one epoch

- Batch gradient descent → 1
- Stochastic gradient descent → N (N = number of data samples)
- Mini-batch gradient descent → N/B (B = batch size

Gradient descent in flat vs steep area





Visualization credit: Niranjan Kumar

Momentum

Go faster if the same direction is repeated

$$v_t = \boxed{\gamma * v_{t-1}} + \eta \nabla w_t$$

$$w_{t+1} = w_t - v_t$$

AdaGrad (adaptive learning rate)

- Parameter that is not zero most of the times -> small learning rate
- Parameter that is zero most of the times, when it is on → higher learning rate to boost the gradient update

$$v_t = v_{t-1} + (\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta}{\boxed{\sqrt{(v_t)}} + \epsilon} \nabla w_t$$

Non-sparse parameters will have large history value due to frequent updates
→ Small learning rate

Root Mean Square Propagation (RMSProp)

Decaying the history to prevent the rapid growth of the denominator in AdaGrad

$$v_t = \boxed{\beta * v_{t-1} + (1 - \beta)(\nabla w_t)^2}$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} \nabla w_t$$

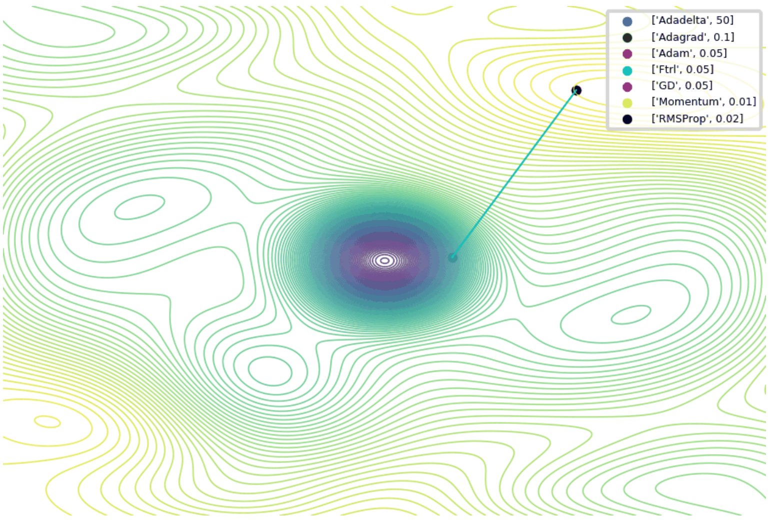
Adaptive moment estimation

- Combine Momentum & RMSProp

$$\boxed{m_t = \beta_1 * v_{t-1} + (1-\beta_1)(\nabla w_t)}$$ → use history to compute update

$$\boxed{v_t = \beta_2 * v_{t-1} + (1-\beta_2)(\nabla w_t)^2}$$ → use history to adjust learning rate (shrink or boost)

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} m_t$$

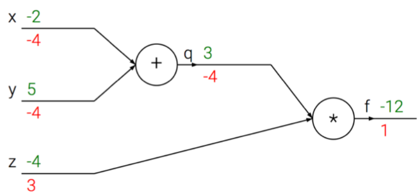


Visualization credit: https://github.com/Jaewan-Yun/optimizer-visualization

Backpropagation

- Forward -> compute loss -> loss run backward to find gradients -> update weights

f (x,y,z) = (x+y) *z




```
# set some inputs
x = -2; y = 5; z = -4

# perform the forward pass
q = x + y # q becomes 3
f = q * z # f becomes -12

# perform the backward pass (backpropagation) in reverse order:
# first backprop through f = q * z
dfdz = q # df/dz = q, so gradient on z becomes 3
dfdq = z # df/dq = z, so gradient on q becomes -4
# now backprop through q = x + y
dqdx = 1.0
dqdy = 1.0
dfdx = dfdq * dqdx # will be z = -4 #And the multiplication here is the chain rule!
dfdy = dfdq * dqdy# will be z = -4 #
print("[dfdx, dfdy, dfdz] = [{}, {}, {}]".format(dfdx, dfdy, dfdz))
```

```
[dfdx, dfdy, dfdz] = [-4.0, -4.0, 3]
```

Credit: Kapathy (cs231n)

Learning hyperparameters

- Learning rate
- Momentum
- Decay
- Epsilon (fuzz factor)
- Beta_1, beta_2 -> Adam

Hyperparameter search with Hyperas

Potential use cases

- Varying dropout probabilities, sampling from a uniform distribution
- Different layer output sizes
- Different optimization algorithms to use
- Varying choices of activation functions
- Conditionally adding layers depending on a choice
- Swapping whole sets of layers

Hyperas' search algorithms

- Random Search
- Tree of Parzen Estimator (TPE)