Course: Computer Vision

Unit 4: Object Recognition

Convolutional Neural Networks (II): Improving the performance

Luis Baumela

Universidad Politécnica de Madrid

Convolutional Neural Networks (II)

1. Introduction
   - Why regularize?
   - Bias-variance tradeoff, Bagging
2. Regularization techniques
   - Data augmentation
   - Weight decay
   - Early stopping
   - Dropout
   - Transfer learning
3. Other performance improvements
   - Batch normalization

4. Homework

Convolutional Neural Networks

- **Acknowledgement**

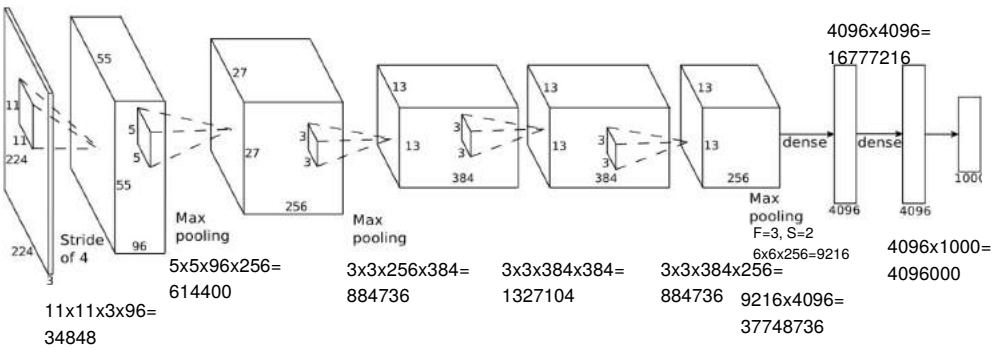
Slides taken from:

- Geoffrey Hinton
- Hugo Larrochelle
- Andrej Karpathy
- Justin Johnson
- ….

Introduction

- Why regularize?

  How many parameters in Alexnet CNN?



ILSVRC: 1.2 M training images

Introduction

- Why regularize?

  How many parameters in Alexnet CNN? **62.3 M**

  How many training images? **1.2 M**

  What will happen if we do not care at all?

  Overfit the training data set → **poor generalization**

  Solution:

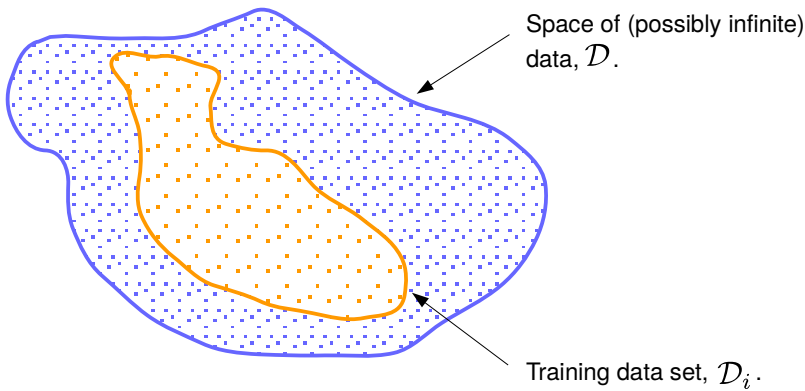
  Reduce the degrees of freedom in your model → **Regularize**

Introduction

- Bias-variance tradeoff

  It emerges when you train a model with a **limited amount of data**.

  The goal is to minimize the **generalization error**, i.e. error on any data either seen (during training) or unseen.



Introduction

- Bias-variance tradeoff

  Let $f(\mathbf{x})$ be the function we want to learn and $g(\mathbf{x}|\mathcal{D}_i)$ the model we learn with data $\mathcal{D}_i$.

  Our expected **generalization error** $\mathcal{E} = \mathbb{E}_{\mathcal{D}_i}[(g(\mathbf{x}|\mathcal{D}_i) - f(\mathbf{x}))^2]$ is

$$\mathcal{E} = \underbrace{\mathbb{E}_{\mathcal{D}_i}[(g(\mathbf{x}|\mathcal{D}_i) - \bar{g})^2]}_{\text{variance}} + \underbrace{(\bar{g} - f(\mathbf{x}))^2}_{\text{bias}^2}$$

  where $\bar{g} = \mathbb{E}_{\mathcal{D}_i}[g(\mathbf{x}|\mathcal{D}_i)]$

  The combination of two terms:

  - Bias. Distance from the average model to the solution.

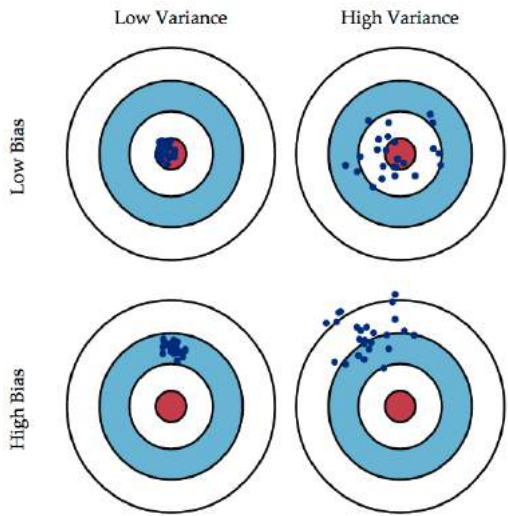
    It is big if the model has too little capacity to fit the data.

  - Variance. Specificity, changes in the model for different training data sets.

    It is big if the model has so much capacity that it also fits the noise.

Introduction

- Bias-variance tradeoff

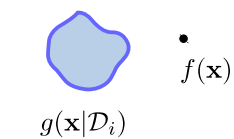


Introduction

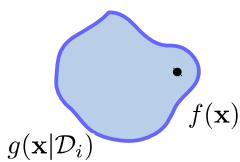
- Bias-variance tradeoff

  It emerges when you train a model with a limited amount of data.

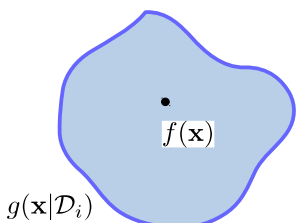
  Changing the number of parameters and checking the total generalization error we may find a trade-off complexity for a certain model.



low variance/high bias



reasonable trade-off



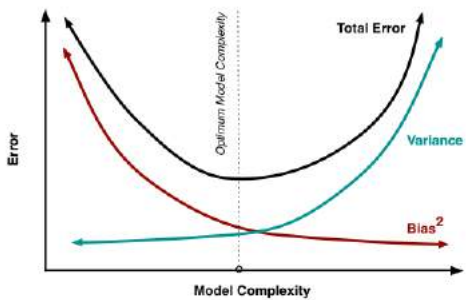
high variance/low bias

Introduction

- Bias-variance tradeoff

  It emerges when you train a model with a limited amount of data.

  Changing the number of parameters and checking the total generalization error we may find a trade-off complexity for a certain model.



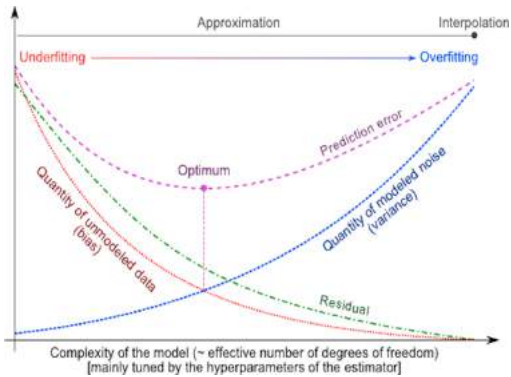
Introduction

- Bias-variance tradeoff

  It emerges when you train a model with a limited amount of data.

  Changing the number of parameters and checking the total generalization error we may find a trade-off complexity for a certain model.

  However, since $f(\mathbf{x})$ is unknown, we cannot compute these curves.



Introduction

- Bias-variance tradeoff

  It emerges when you train a model with a limited amount of data.

  Changing the number of parameters and checking the total generalization error we may find a trade-off complexity for a certain model.

  We can estimate this trade-off from the observation that

  - Underfit models tend to have the similar **poor** goodness of fit performance **both on training and testing data sets**.
  - Overfit models tend to have much **worse** goodness of fit performance **on a testing dataset** vs. a training data set.

Introduction

- Bias-variance tradeoff

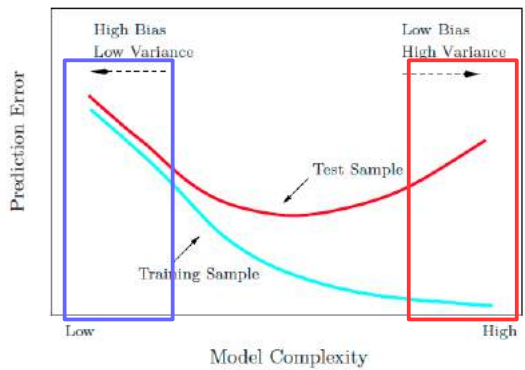
  It emerges when you train a model with a limited amount of data.

  Changing the number of parameters and checking the total generalization error we may find a trade-off complexity for a certain model.

  We can estimate this trade-off from

High test & training errors

Underfitting



High test & low training errors

Overfitting

Introduction

- Bias-variance tradeoff

  It emerges when you train a model with a limited amount of data.

  Changing the number of parameters and checking the total generalization error we may find a trade-off complexity for a certain model.

  What to do if we have a model with:

  - High variance:
    - Increase the data set
    - Reduce the number of parameters (net layers, cell units, weights).
    - Increase regularization
  - High bias:
    - Increase the number of parameters (net layers, cell units, weights).
    - Reduce regularization.

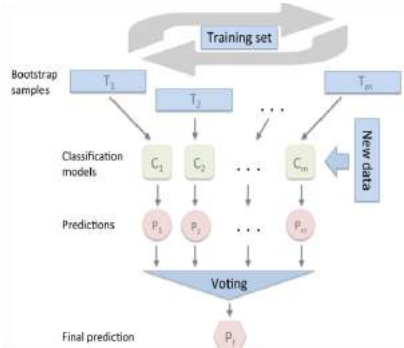
Introduction

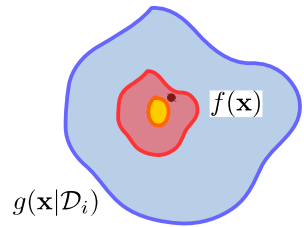
- Bias-variance tradeoff, Bagging

  An alternative approach is to “average” several high variance/low bias models.

Bagging (Bootstrap Aggregation)

Drawing *m* Bootstrap samples from $\mathcal{D}_i$





lower variance
with bagging

Bagging reduces the variance:

$$\mathcal{E} = \underbrace{\mathbb{E}_{\mathcal{D}_i}[(g(\mathbf{x}|\mathcal{D}_i) - \bar{g})^2]}_{\text{variance} \to 0} + \underbrace{(\bar{g} - f(\mathbf{x}))^2}_{\text{bias}^2}$$

Regularization

- Data augmentation

  Scarce data is the main bottleneck in deep models.

  - CNNs have some built-in invariances:

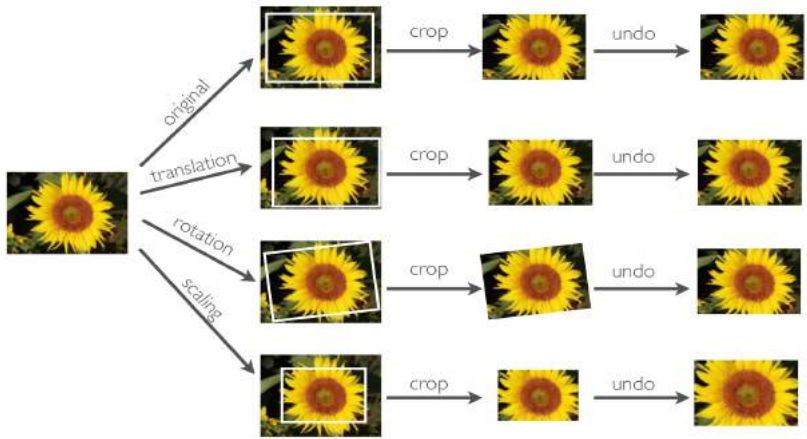
    small translations due to convolution and max pooling

  - Not invariant to other important variations such as rotations, scale, colour changes, noise, etc..

  - However, it's easy to artificially generate data with such transformations
    - ➔ use such data as additional training data
    - ➔ NN will learn to be invariant to such transformations

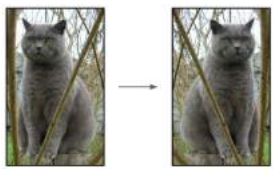
Regularization

- Data augmentation (different pixels, same label)
  - Translation, rotation, scale, shear changes:



Regularization

- Data augmentation (different pixels, same label)
  - Translation, rotation, scale, shear changes
  - Horizontal flips



  - Shuffle RGB channles

Regularization

- Data augmentation (different pixels, same label)
  - Translation, rotation, scale, shear changes
  - Horizontal flips
  - Add jitter contrast
  - Add ‘‘elastic’’ deformations (useful in character recognition)
    applying a ‘‘distortion field’’ that specifies where to displace each pixel

- At testing

  Average network response at a fixed set of tranformations

Regularization

- Data augmentation (different pixels, same label)

Some support from Keras

```
keras.preprocessing.image.ImageDataGenerator(featurewise_center=False, samplewise_center
```

- **featurewise_center**: Boolean. Set input mean to 0 over the dataset, feature-wise.
- **samplewise_center**: Boolean. Set each sample mean to 0.
- **featurewise_std_normalization**: Boolean. Divide inputs by std of the dataset, feature-wise.
- **samplewise_std_normalization**: Boolean. Divide each input by its std.
- **zca_epsilon**: epsilon for ZCA whitening. Default is 1e-6.
- **zca_whitening**: Boolean. Apply ZCA whitening.
- **rotation_range**: Int. Degree range for random rotations.
- **width_shift_range**: Float, 1-D array-like or int
- **horizontal_flip**: Boolean. Randomly flip inputs horizontally.
- **vertical_flip**: Boolean. Randomly flip inputs vertically.
- **rescale**: rescaling factor. Defaults to None. If None or 0, no rescaling is applied, otherwise we multiply the data by the value provided (after applying all other transformations).
- **preprocessing_function**: function that will be implied on each input. The function will run after the image is resized and augmented. The function should take one argument: one image (Numpy tensor with rank 3), and should output a Numpy tensor with the same shape.

Regularization

- Data augmentation (different pixels, same label)

Some support from Keras

```
keras.preprocessing.image.ImageDataGenerator(featurewise_center=False, samplewise_center
```

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
y_train = np_utils.to_categorical(y_train, num_classes)
y_test = np_utils.to_categorical(y_test, num_classes)

datagen = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True)

# compute quantities required for featurewise normalization
# (std, mean, and principal components if ZCA whitening is applied)
datagen.fit(x_train)

# fits the model on batches with real-time data augmentation:
model.fit_generator(datagen.flow(x_train, y_train, batch_size=32),
                    steps_per_epoch=len(x_train) / 32, epochs=epochs)
```

Regularization

- Weight decay

  Let $E(\mathbf{w})$ be the loss function minimized in the NN.

  We modify the weights in the steepest descent direction:

  $$w_i \leftarrow w_i - \eta \frac{\partial E(\mathbf{w})}{\partial w_i}$$

  where η is the learning rate.

  We may limit the number of free parameters by introducing a zero mean Gaussian prior over the weights:

  $$\tilde{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^2$$

  the steepest descent algorithm

  $$w_i \leftarrow w_i - \eta \frac{\partial E(\mathbf{w})}{\partial w_i} - \eta\lambda w_i$$

  where λ is the “weight decay” regularization parameter.

Regularization

- Weight decay

  We modify the weights in the steepest descent direction:

$$w_i \leftarrow w_i - \eta \frac{\partial E(\mathbf{w})}{\partial w_i} - \eta \lambda w_i$$

  The regularization term $\eta \lambda w_i$ causes the weight to decay in proportion to its size.

  In practice this penalizes large weights and effectively limits the freedom in the model.

  - In Keras

The penalties are applied on a per-layer basis. The exact API will depend on the layer, but the layers `Dense`, `Conv1D`, `Conv2D` and `Conv3D` have a unified API.

```
from keras import regularizers
model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l2(0.01),
```

Regularization

- Early stopping

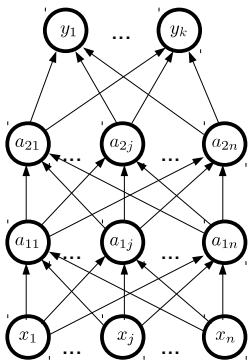
  Training with no weight decay and “stopping early” the minimization is equivalent to using a quadratic weight decay.

Regularization

- Dropout

  Prune the NN by ramdomly discarding hidden units

  The output of each hidden unit is multiplied by “0” with a probability p (for example p=0.5)

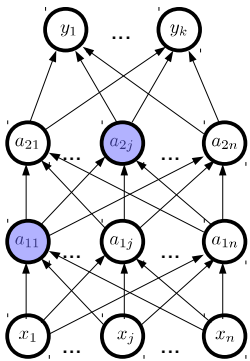


Regularization

- Dropout

  Prune the NN by ramdomly discarding hidden units

  The output of each hidden unit is multiplied by “0” with a probability p (for example p=0.5)



Regularization

- Dropout

  Prune the NN by ramdomly discarding hidden units

  The output of each hidden unit is multiplied by “0” with a probability p (for example p=0.5). Now

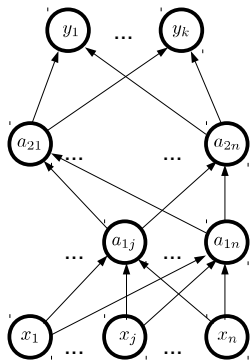
  - Hidden units cannot co-adapt
  - Hidden units must learn more general features

  Changes on the **training algorithm**:

  - Forwprop. Each layer includes a random [0,1] mask that multiplies each activation.
  - Backprop. Hidden units multiplied by “0” do not contribute to the gradient backprop.

  Changes on **testing**:

  - Masks are replaced by their expectations.



Regularization

- Dropout, comments

  The SGD with dropout works slightly differently:

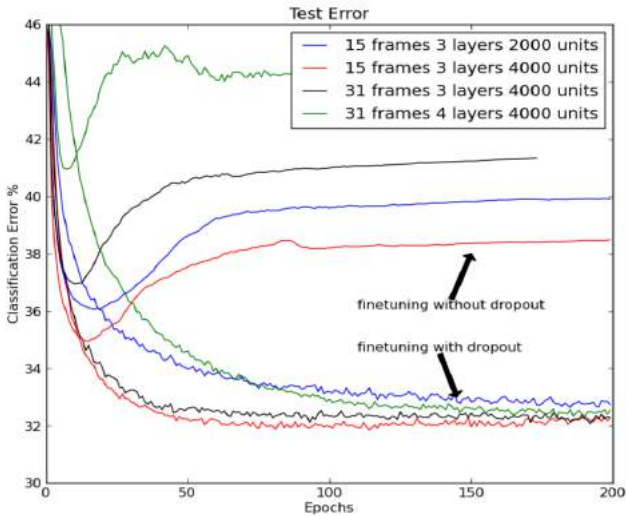
  - SGD usually works best with a small learning rate that results in a smoothly decreasing objective function.
  - Dropout works best with a larger learning rate, resulting in a constantly fluctuating objective function
  - SGD moves slowly and steadily in the most promising direction
  - Dropout rapidly explores many different directions and rejects the ones that worsen performance.

  Training with dropout can be slower, but it pays off!

  Very useful in fully connected layers. Also CNN ones.

Regularization

- Dropout, results



Regularization

- Dropout, in Keras

```
keras.layers.Dropout(rate, noise_shape=None, seed=None)
```

Applies Dropout to the input.

Dropout consists in randomly setting a fraction `rate` of input units to 0 at each update during training time, which helps prevent overfitting.

Arguments

- **rate**: float between 0 and 1. Fraction of the input units to drop.
- **noise_shape**: 1D integer tensor representing the shape of the binary dropout mask that will be multiplied with the input. For instance, if your inputs have shape `(batch_size, timesteps, features)` and you want the dropout mask to be the same for all timesteps, you can use `noise_shape=(batch_size, 1, features)`.
- **seed**: A Python integer to use as random seed.

Regularization

- Transfer learning (pre-training)

  What to do if we have a problem with a small data set.

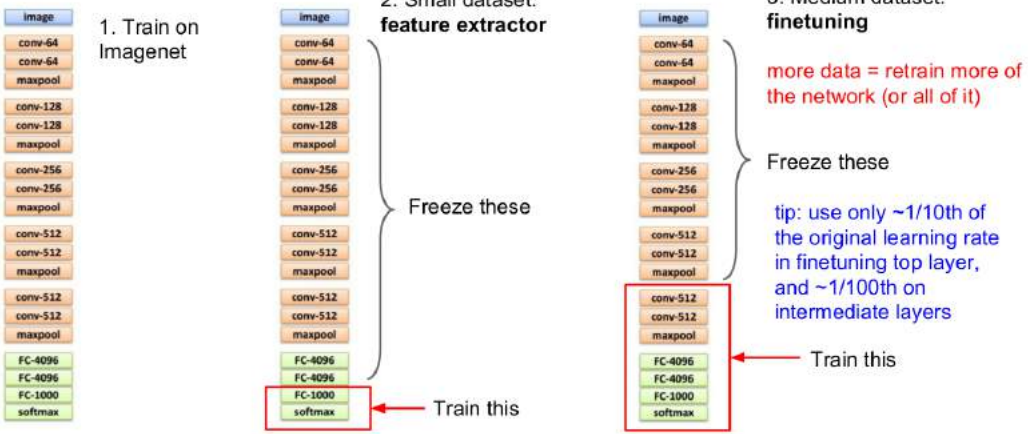
  1. Use another (similar) large data set to train a large CNN.
  2. Re-train the large CNN with your small data set.

  Take advantage of

  - Keras applications package
  - pyTorch, Tensor Flow, Caffe model zoo models in the net
  - Follow procedures to convert from one model to another

Regularization

- Transfer learning (pre-training)



Regularization

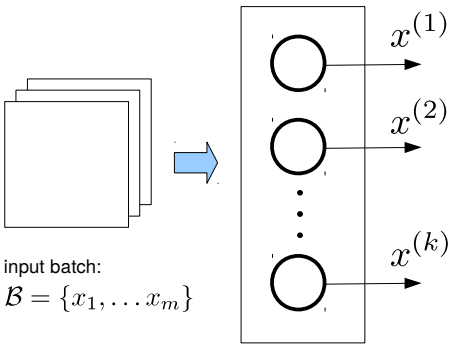
- Transfer learning in Keras

```
from keras.applications.inception_v3 import InceptionV3
from keras.preprocessing import image
from keras.models import Model
from keras.layers import Dense, GlobalAveragePooling2D
from keras import backend as K

# create the base pre-trained model
base_model = InceptionV3(weights='imagenet', include_top=False)

# add a global spatial average pooling layer
x = base_model.output
x = GlobalAveragePooling2D()(x)
# let's add a fully-connected layer
x = Dense(1024, activation='relu')(x)
# and a logistic layer -- let's say we have 200 classes
predictions = Dense(200, activation='softmax')(x)

# this is the model we will train
model = Model(inputs=base_model.input, outputs=predictions)

# first: train only the top layers (which were randomly initialized)
# i.e. freeze all convolutional InceptionV3 layers
for layer in base_model.layers:
    layer.trainable = False

# compile the model (should be done *after* setting layers to non-trainable)
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

# train the model on the new data for a few epochs
model.fit_generator(...)
```

Other performance improvements

- Batch normalization

  Stantardizes the inputs to all activation units for each training batch.

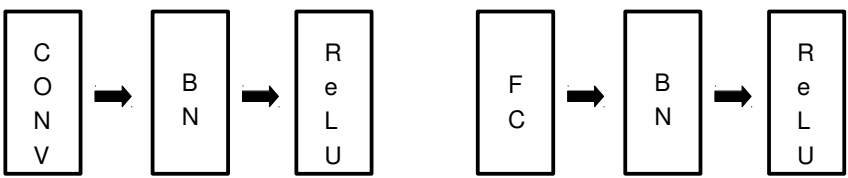


Normalizes:

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_{\mathcal{B}^{(k)}}}{\sigma_{\mathcal{B}^{(k)}}}$$

Learns:

$$y_i^{(k)} = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)}$$



Other performance improvements

- Batch normalization

  Stantardizes the inputs to all activation units for each training batch.

Training:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Testing:

During training learns

- running:
  - → means and,
  - → variances of activations

or

- Post-training mean and variances.

At testing uses fixed learned means and variances of activations.

Other performance improvements

- Batch normalization

  Stantardizes the inputs to all activation units for each training batch.

  Discussion:

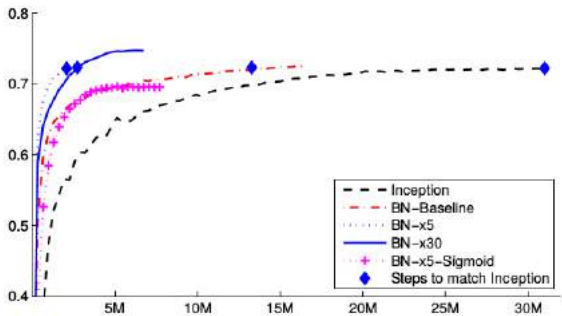
  - Network can learn
    $$\gamma = \sigma_{\mathcal{B}}; \quad \beta = \mu_{\mathcal{B}}$$
    so the identiy may be represented.
  - Improves optimization (gradients do no vanish).
  - Allows higher learning rates.
  - Reduces dependence on inicialization
  - Regularizes

Other performance improvements

- Batch normalization

  Stantardizes the inputs to all activation units for each training batch.

  Results for training the Inception module with different configurations:



Other performance improvements

- Batch normalization in Keras

```
keras.layers.BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001, center=True, scal
```

Normalize the activations of the previous layer at each batch, i.e. applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1.

Arguments

- **axis**: Integer, the axis that should be normalized (typically the features axis). For instance, after a `Conv2D` layer with `data_format="channels_first"`, set `axis=1` in `BatchNormalization`.
- **momentum**: Momentum for the moving mean and the moving variance.
- **epsilon**: Small float added to variance to avoid dividing by zero.
- **center**: If True, add offset of `beta` to normalized tensor. If False, `beta` is ignored.
- **scale**: If True, multiply by `gamma`. If False, `gamma` is not used. When the next layer is linear (also e.g. `nn.relu`), this can be disabled since the scaling will be done by the next layer.
- **beta_initializer**: Initializer for the beta weight.
- **gamma_initializer**: Initializer for the gamma weight.
- **moving_mean_initializer**: Initializer for the moving mean.
- **moving_variance_initializer**: Initializer for the moving variance.

Homework for the next two weeks

Goal:

Improve the performance of your NN by regularizing and processing the data through the net.

Steps:

- Use any combination of regularization or data processing procedures to improve the performance of both, the ffNN and CNN, that you have built.

Submission deadline: Dec, 28th, 2018