Convolutional Neural Networks and how to fit them

Eilif Solberg

TEK5040/TEK9040

Outline

Supervised Deep Learning Cheat Sheet

1. Generate training and validation data on the form x, y, where y is the supervision, e.g. class label, only available during training. Later, add data augmentation to training data to improve generalization.
2. Create a suitable model (use pretrained model if appropriate). Add model regularization to improve generalization.
3. Define a metric (what you care about) and loss function (something differentiable).
4. Choose an optimizer (SGD-like) and learning rate (schedule).
5. Define training step where we on a minibatch x, y:
   - Calculate model predictions $\hat{y}$ and loss wrt y.
   - Find gradient of loss with respect to variables
   - Update model paramaters either by passing gradients to optimizer, or by using custom update rule.
   - Update loss and metric summaries.
6. Define validation step where we on each iteration update metric (and optionally loss) summaries.
7. Regularly plot loss and metric as well as other potential summaries.

Neural Networks

Biological model

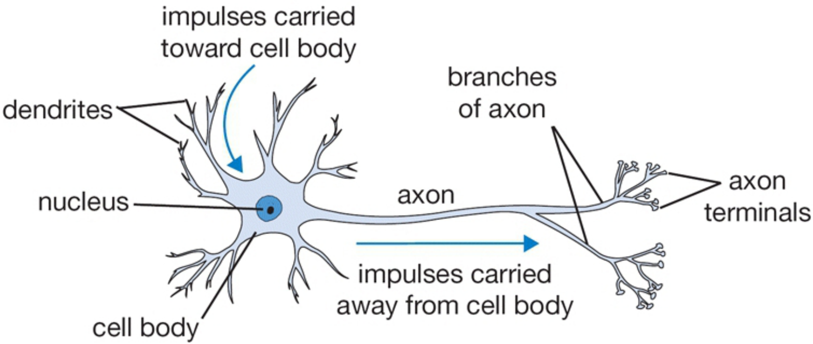


Figure: Biological model of neuron. Illustration from
http://cs231n.github.io/neural-networks-1/

Mathematical model

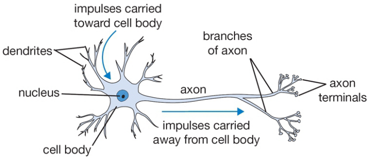


Figure: Biological model of neuron.
Illustration from
http://cs231n.github.io/neural-networks-1/

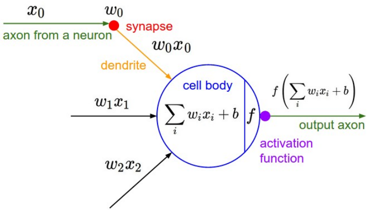


Figure: Mathematical model of neuron. Illustration from
http://cs231n.github.io/neural-networks-1/

Neuron in TensorFlow

```python
import numpy as np
import tensorflow as tf

class Neuron(object):
  def __init__(self, num_inputs):
    self.weights = tf.Variable(np.random.uniform(-0.1, 0.1, size=num_inputs)) # random uniform initialzation
    self.bias = tf.Variable(0) # zero initialization

  def __call__(self, x):
    """Calculate activation for the neuron."""
    cell_body_sum = tf.reduce_sum(x*self.weights) + self.bias
    # apply sigmoid activation function
    firing_rate = 1.0 / (1.0 + tf.exp(-cell_body_sum))
    return firing_rate

neuron = Neuron(num_inputs=5)
output = neuron([0.1, 0.4, -0.3, 0.7, -1.3])
```

- Why do we define the weights and bias with `tf.Variable`?

Detector / activation function

- Non-saturating activation functions as ReLU, leaky ReLU dominating

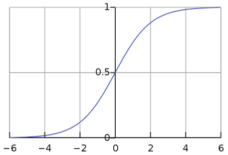


Figure: Sigmoid function

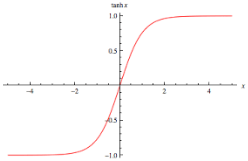


Figure: Tanh function

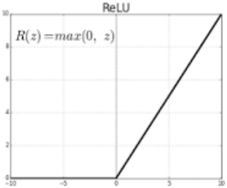


Figure: ReLU function

Activation functions in TensorFlow

Some commonly used activations functions are already implemented and can be found at `tf.keras.activations`, e.g.

```
# Note that activation functions work elementwise on the input
↪ tensor/array
tf.keras.activations.relu # f(x) = tf.maximum(x, 0)
tf.keras.activations.tanh # f(x) =
↪ (tf.exp(2*x)-1)/(tf.exp(2*x)+1)
tf.keras.activations.sigmoid # f(x) = 1 / (1+tf.exp(-x))
```

Note: activation functions with *trainable parameters* are found under `tf.keras.layers` and start with *uppercase* letters.

Network with several fully connected layers

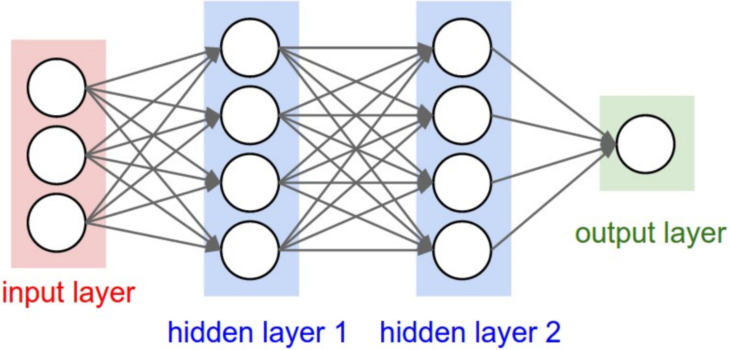


Figure: Illustration from http://cs231n.github.io/neural-networks-1/

- What is the number of parameters?
- Mathematican: One hidden layer is enough

Layer of neurons in TensorFlow

```
class FullyConnected_v1(object):
  def __init__(self, num_inputs, num_outputs):
    self.weights = tf.Variable(np.random.uniform(-0.1, 0.1, size=[num_inputs, num_outputs]))
    self.bias = tf.Variable(np.zeros(num_outputs))

  def __call__(self, x):
    # Why do we right-multiply with matrix rather than left-multiply?
    return tf.matmul(x, self.weights) + self.bias

# array of shape [batch_size, 3] ==> [batch_size, 5]
fc = FullyConnected_v1(nun_inputs=3, num_outputs=5)
# array of shape [2, 3] ==> [2, 5]
fc(np.array([[1.0, 0.4, 0.2], [-0.4, 0.3, 0.2]]))
```

Layer of neurons in TensorFlow as Keras Layer

```python
class FullyConnected_v2(tf.keras.layers.Layer):
  def __init__(self, num_outputs):
    super(FullyConnected_v2, self).__init__()
    self.num_outputs = num_outputs

  def build(self, input_shape):
    """Assume input_shape[0] is batch size and input_shape[1] is size of input samples."""
    self.W = tf.Variable(np.random.uniform(-0.1, 0.1, size=[input_shape[1], self.num_outputs]))
    self.b = tf.Variable(np.zeros(self.num_outputs))

  def call(self, x):
    return tf.matmul(x, self.W) + self.b

# array of shape [batch_size, num_inputs] ==> [batch_size, 5]
fc = FullyConnected_v2(num_outputs=5)
# array of shape [2, 3] ==> [2, 5]
fc(np.array([[1.0, 0.4, 0.2], [-0.4, 0.3, 0.2]]))
```

- Why don't we need to specify the number of input nodes?
- Can we later give an array of shape $[2, 4]$ as input?

Notes on extending Keras layer

- The `build` method is called the first time `__call__` is called.
- We implement `call` rather than `__call__`. The `__call__` metods is implemented in the parent class, and will call 'call'
- We may also use `self.add_variable` method to add variables to layer
- Remember to call `super` method in `__init__` to initialize `Layer` class properly.
- This layer can already be found at `tf.keras.layers.Dense` (with more functionality)
- Optional: Implement `get_config` and `compute_output_shape` (for serialization and model summary purposes respectively).
- See custom layers and models guide and Layer documentation.

Create model

```python
class MyModel(tf.keras.Model):
  def __init__(self):
    super(MyModel, self).__init__()
    self.d1 = tf.keras.layers.Dense(4, activation='relu')
    self.d2 = tf.keras.layers.Dense(4, activation='relu')
    self.d3 = tf.keras.layers.Dense(1, activation='sigmoid')

  def call(self, x):
    x = self.d1(x)
    x = self.d2(x)
    return self.d3(x)

model = MyModel()
print(model(np.array([[1.0, 0.4, 0.2], [-0.4, 0.3, 0.2]])))
print(model.summary())
```

Use `kernel_initializer` and `bias_initializer` arguments to specify initialization scheme different from default.

- Will later look at simpler ways as well to create a model.

CNN Architectures

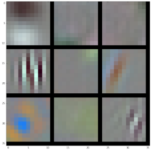
Template matching

Figure: Illustration from
http://pixuate.com/technology/template-matching/

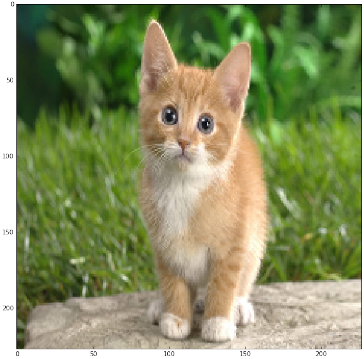
1. Try to match template at each location by "sliding over window"
2. Threshold for detection

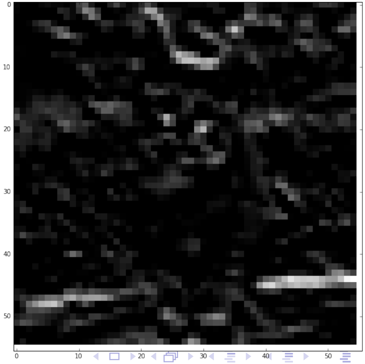
For 2D-objects, kind of possible but difficult

Convolution



Which filter has produces the activation map on the right?





Convolutional layer

–> Glorified template matching

- Many templates (aka output filters)
- We *learn* the templates, the *weights* are the templates
- Intermediate detection results only *means to an end*
    - treat them as *features*, which we again match new templates to
- Starting from the second layer we have "nonlinear filters"

Hyperparameters of convolutional layer

1. Kernel height and width - template sizes
2. Stride - skips between template matches
3. Dilation rate
   - *Holes* in template where we "don't care".
   - Larger field-of-view without more weights...
4. Number of output filters - number of templates
5. Padding - expand image, typically with zeros

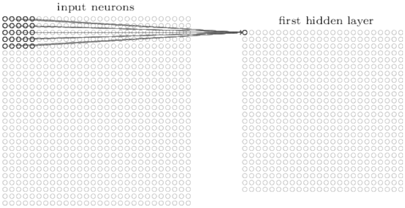


Figure: Image from http://neuralnetworksanddeeplearning.com/

2D-Convolutional layers in TensorFlow

```
conv = tf.keras.layers.Conv2D(
  filters, # e.g. 64
  kernel_size, # e.g. (3, 3)
  strides=(1, 1),
  padding='valid', # other option is 'same'
  data_format=None,
  dilation_rate=(1, 1),
  activation=None,
  use_bias=True,
  kernel_initializer='glorot_uniform',
  bias_initializer='zeros',
  kernel_regularizer=None,
  bias_regularizer=None,
  activity_regularizer=None,
  kernel_constraint=None,
  bias_constraint=None,
  **kwargs)
```

Note: some arguments can be both *strings* and *python objects*.

Pro tip

Many properties of convolutional layers can be most easily studied by considering 1D convolutions.

1. Shape of output (P=padding, W=kernel_width, S=stride)

$$(input\_width + 2P - W)/S + 1$$

2. Field-Of-View as function of depth d (if stacked), assuming no stride, and W odd.

$$d(W - 1) + 1$$

Basic CNN architecture for image classification

Image –> [Conv –> ReLU]xN –> Fully Connected –> Softmax

- Increase filter depth when using stride

Improve with:

- Batch normalization
- Skip connections ala ResNet or DenseNet
- No fully connected, average pool predictions instead

CNN TensorFlow

Can use `tf.keras.models.Sequential` to easily define a model from a sequence of layers

```
from tensorflow.keras.layers import Conv2D, Flatten, Dense

model = tf.keras.models.Sequential([
  Conv2D(32, kernel_size=3, activation='relu'),
  Conv2D(64, kernel_size=3, strides=2, activation='relu'),
  Flatten(),
  Dense(128, activation='relu'),
  Dense(10, activation='softmax')
])
```

CNN TensorFlow II

```
inputs = tf.keras.Input(shape=(32, 32, 3))
x = Conv2D(32, kernel_size=3, activation='relu')(inputs)
x = Conv2D(64, kernel_size=3, strides=2, activation='relu')(x)
x = Flatten()(x)
x = Dense(128, activation='relu')(x)
outputs = Dense(10, activation='softmax')(x)

model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

- Allows for non-sequential structure, but structure of layers is still fixed in advance.

CNN TensorFlow III

```
class MyModel(tf.keras.Model):
  def __init__(self):
    super(MyModel, self).__init__()
    self.conv1 = Conv2D(32, kernel_size=3, activation='relu')
    self.conv2 = Conv2D(64, kernel_size=3, strides=2, activation='relu')
    self.flatten = Flatten()
    self.d1 = Dense(128, activation='relu')
    self.d2 = Dense(10, activation='softmax')

  def call(self, x):
    x = self.conv1(x)
    x = self.conv2(x)
    x = self.flatten(x)
    x = self.d1(x)
    return self.d2(x)
model = MyModel()
```

- Most flexible, but more code (and thus room for mistakes).
- Need model code to restore model as python object.

Fitness

How do we fit model?

How do we find parameters θ for our network?

Supervised learning

- Training data comes as (X, Y) pairs, where Y is the target
- Want to learn $f_\theta(x) \sim p(y|x)$, conditional distribution of Y given X, where θ are our parameters.
- Define *differentiable* surrogate loss function, e.g. for a single sample with $Y \in \mathbb{R}^n$ and $Y \in \mathbb{N}$ respectively:

$$l(\theta) = l(f_\theta(X), Y) = \sum_{i=1}^{n} ((f_\theta(X))_i - Y_i)^2 \qquad \text{squared error loss}$$

$$l(\theta) = l(p_\theta(X), Y) = -log(p_\theta(X)_Y) \qquad \text{negative likelihood}$$

The first loss is common in *regression*, while the second is common in *classification*.

Losses in TensorFlow

```python
def mean_squared_error(y_true, y_pred):
  """y_true: [batch_size, n], y_pred : [batch_size, n]"""
  # sum over axis, mean over batch dimension
  return tf.reduce_mean(tf.reduce_sum((y_true-y_pred)**2, axis=-1))

def sparse_categorical_cross_entropy(y_true, y_pred):
  """y_true: [batch_size], y_pred : [batch_size, num_classes]"""
  y_true = tf.expand_dims(y_true, axis=-1) # [batch_size] ==> [batch_size, 1]
  # tf.gather --> extracts probabilities from y_pred using indices in y_true
  log_likelihoods = tf.math.log(tf.gather(y_pred, y_true, batch_dims=1))
  return -tf.reduce_mean(log_likelihoods)

mean_squared_error(np.zeros((3, 5)), np.ones((3, 5))) # ==> 5
sparse_categorical_cross_entropy([0, 1, 2], [[.9, .05, .05], [.5, .89, .6], [.05, .01, .94]])
```

Losses in TensorFlow II

Many losses can already be found implemented at `tf.keras.losses`.

```
mse = tf.keras.losses.MeanSquaredError()
loss = mse([0., 0., 1., 1.], [1., 1., 1., 0.])
print('Loss: ', loss.numpy())  # Loss: 0.75

cce = tf.keras.losses.SparseCategoricalCrossEntropy(from_logits=False) # set true if not softmax applied
loss = cce([0, 1, 2], [[.9, .05, .05], [.5, .89, .6], [.05, .01, .94]])
print('Loss: ', loss.numpy())  # Loss: 0.3239
```

Optimization

Gradient

- For a function $f : \mathbb{R}^n \to \mathbb{R}$ the *gradient* is the n-dimensional vector of all partial derivatives of the f with respect to the input variables.
- The gradient is the direction for which the function increases the most.

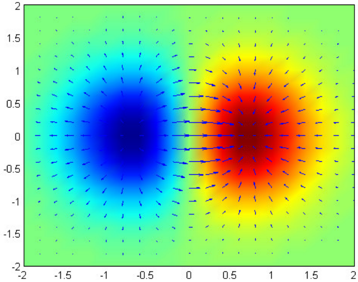


Figure: Gradient of the function $f(x^2, y^2) = x/e^{x^2+y^2}$ [By Vivekj78 [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)], from Wikimedia Commons]

How do we find the gradient?

- Approximate by finite differences. Recall that for a function of one variable

$$\frac{d}{dx}f(x) \approx \frac{f(x+h) - f(x)}{h}$$

  for small enough h. How does this scale with number of variables?
- Analytically with backpropagation
  - Integration is an art - derivation is craftmanship.
  - Gradients propagated from output towards input.

Automatic differentiation - first order

Compute first order derivatives

```
x = tf.constant(3.0)
with tf.GradientTape() as g:
  g.watch(x) # keep 'tape' of values that x affects
  y = x * x
# Find derivative of y with respect to x
dy_dx = g.gradient(y, x) # Will compute to 6.0 (dy_dx x^2 = 2x)
```

Note that y should always be a scalar (typically our loss value), while x can in general be a vector (typically the parameters of our model).

Automatic differentiation wrt. variables

No need to explicitly add 'watch' for *trainable variables*

```
x = tf.Variable(3.0, trainable=True) # Note: 'trainable' is
↪ True by default
with tf.GradientTape() as g:
    y = x * x
dy_dx = g.gradient(y, x) # Will compute to 6.0 (dy_dx x^2 = 2x)
```

Automatic differentiation - second order

Compute first and second order derivatives

```python
x = tf.Variable(3.0)
with tf.GradientTape() as g:
  with tf.GradientTape() as gg:
    y = x * x
  dy_dx = gg.gradient(y, x) # Will compute to 6.0 (dy_dx x^2 = 2x)
d2y_dx2 = g.gradient(dy_dx, x) # Will compute to 2.0 (dy_dx 2x = 2)
```

For more information:

- https://www.tensorflow.org/api_docs/python/tf/GradientTape

(Stochastic) gradient descent

Taking steps in the opposite direction of the gradient

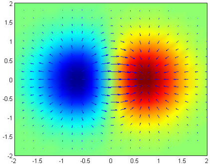


Figure: [By Vivekj78 [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)], from Wikimedia Commons]

- Full gradient too expensive / not necessary

$$\sum_{i=1}^{N} \nabla_\theta l(f(X_i), Y_i) \approx \sum_{i=1}^{n} \nabla_\theta l(f(X_{P(i)}), Y_{P(i)}) \tag{1}$$

for a random permutation P.

Updating model with vanilla gradient descent

```
for x, y in train_data:
  with tf.GradientTape() as tape:
    y_pred = model(x, training=True) # training argument only needed when model has different behaviour under training and inference
    loss = loss_fn(y, y_pred)

  grads = tape.gradient(loss, model.trainable_variables)
  for grad, var in zip(grads, model.trainable_variables):
    var.assign_add(-lr*grad) # var = var - lr*grad
```

Can we do better than basic gradient descent with fixed step size?

Updating model with optimizer

'Optimizers' try to improve upon the simple update rule above by e.g. trying to incorporate some kind of curvature (without calculating second derivatives!)

```
optimizer = tf.keras.optimizers.SGD(0.0001, momentum=0.9)
for x, y in train_data:
  with tf.GradientTape() as tape:
    y_pred = model(x, training=True)
    loss = loss_fn(y, y_pred)

  grads = tape.gradient(loss, model.trainable_variables)
  optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Optimizer

```
print(tf.keras.optimizers.Adam.__doc__)
```

```
Optimizer that implements the Adam algorithm. Adam optimization is a
stochastic gradient descent method that is based on adaptive
estimation of first-order and second-order moments. According to the
paper [Adam: A Method for Stochastic Optimization. Kingma et al.,
2014](http://arxiv.org/abs/1412.6980), the method is
"*computationally efficient, has little memory requirement, invariant
 to diagonal rescaling of gradients, and is well suited for problems
that are large in terms of data/parameters*".

For AMSGrad see [On The Convergence Of Adam And Beyond.
Reddi et al., 5-8](https://openreview.net/pdf?id=ryQu7f-RZ).
```

- SGD with momentum, RMSprop, Adam are popular choices
- For more see https://www.tensorflow.org/api_docs/python/tf/keras/optimizers.

Learning rate schedule

Normally you want to reduce your learning rate as training progresses (typically when loss stops decreasing).

```
class MySchedule(tf.keras.optimizers.
    schedules.LearningRateSchedule):
  def __call__(self, step):
    if step < 100000:
      lr = 0.1
    elif 100000 <= step < 200000:
      lr = 0.01
    else:
      lr = 0.001
    return lr

optimizer = tf.keras.optimizers.SGD(M
    ySchedule())
```

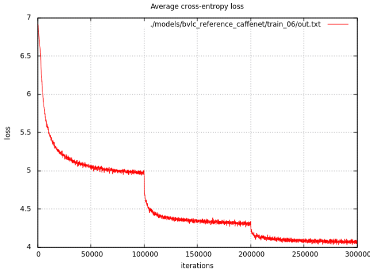


Figure: Example train run following learning rate schedule shown left.

See e.g. `tf.keras.optimizers.schedules` for common learning rate schedules.

Regularization

Overfitting

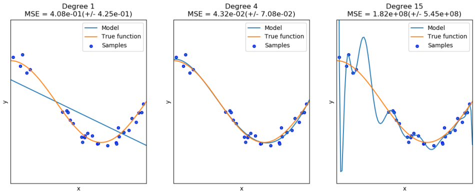


Figure: Model complexity. {Image from scikit-learn}

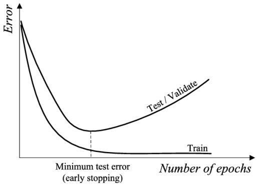


Figure: Train vs test error

- Early stopping is an option, but can we do better?
- Also see tutorial https://www.tensorflow.org/tutorials/keras/overfit_and_underfit

Regularization vs optimization

- Optimization: try to reduce *training loss*
  - often leads to reduction of validation/test loss as a side-effect
- Regularization: try to reduce *test loss*
  - may lead to increase in train loss

Weight regularization

- Penalize non-smooth functions by penalizing large values for the model weights.
- Weight penalty added to loss term, usually squared L2 normalization uniformly for all parameters

$$J(\theta) = l(\theta) + \lambda \|\theta\|_2^2$$

where $\lambda \geq 0$. In TensorFlow this might look like

```
v = tf.Variable([[0.0, 1.0, 2.0], [0.0, -1.0, -2.0]])
for x, y in train_data:
  with tf.GradientTape() as tape:
    y_pred = tf.matmul(x, v)
    loss = loss_fn(y, y_pred) # primary loss
    loss += 0.0001*tf.reduce_sum(v**2) # first iter: 0.0001*10 = 0.001
  # ... compute gradient, update model
```

Weight regularization in TensorFlow - layer

```python
dense = tf.keras.layers.Dense(10, kernel_regularizer=tf.keras.regularizers.l2(0.0001))
# each time dense is run for input x, a loss is added to dense.losses
for x, y in train_data:
  with tf.GradientTape() as tape:
    y_pred = dense(x)
    loss = loss_fn(y, y_pred) # primary loss
    loss += sum(dense.losses) # add regularization loss
    assert dense.losses[0].numpy() == (0.0001*tf.reduce_sum(dense.kernel**2)).numpy()
  # ... compute gradient, update model
```

Weight regularization in TensorFlow - use in model

If you have a `tf.keras.Model` object, *model*, the losses for all layers will be collected into the list *model.losses*.

```
for x, y in train_data:
  with tf.GradientTape() as tape:
    y_pred = model(x, training=True)
    loss = loss_fn(y, y_pred) # primary loss
    loss += sum(model.losses) # add regularization loss for all layers
  # ... compute gradient, update model
```

Dropout

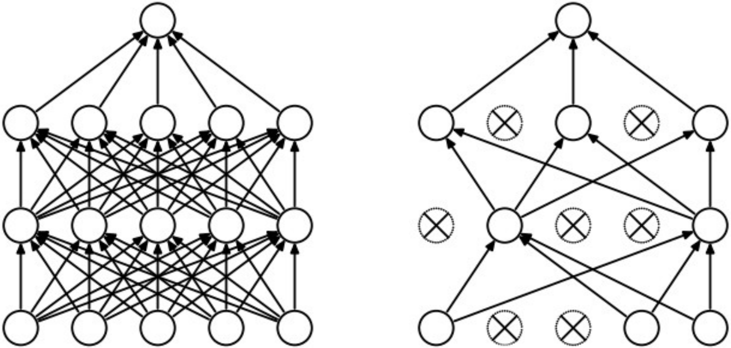


Figure: Left: Inference execution of model. Right: sample of train execution.

TensorFlow dropout

```
dropout = tf.keras.layers.Dropout(0.5)
tf.random.set_seed(123)
x = [[1.0, 1.0, 1.0], [1.0, 1.0, 1.0]]
dropout(x, training=True)  # [[0., 2., 0.], [2., 2., 2.]]
dropout(x, training=True)  # [[2., 0., 2.], [2., 2., 0.]]
dropout(x, training=False) # [[1., 1., 1.], [1., 1., 1.]]
```

- Different behaviours during training and inference (randomness only during training)
- *Expected value* remains the same

Batch normalization

- Unit normalize the input of a neuron, or set of (related) neurons, over the batch.
- Idea: keep mean and standard deviation of input fairly constant to improve optimization.
  - Many theories why it works.
- Turns out randomness also act as regularization.
- Your best friend and your worst enemy

TensorFlow batch normalization

```python
batch_norm = tf.keras.layers.BatchNormalization(axis=-1, center=False, scale=False)
x = np.array([[-1.0, 4.0, 1.0], [1.0, -4.0, 3.0]])
# [batch_size, num_features] == [2, 3]
# feature 1: mean = 0, std ~= 1
# feature 2: mean = 0, std ~= 4
# feature 3: mean = 2, std ~= 1
batch_norm(x, training=True) # ~= [[-1.0, 1.0, -1.0], [1.0, -1.0, 1.0]]
batch_norm(x, training=False) # ~= ?
```

- Different behaviours during training and inference (randomness only during training)
- *Expected value* about the same

Batch normalization for image data

For a tensor [batch_size × height × width × depth], normalize "template matching scores" for each template d by

$$\mu_d \leftarrow \frac{1}{N*H*W}\sum_{i=1}^{N}\sum_{h=1}^{H}\sum_{w=1}^{W} x_{i,h,w,d} \tag{2}$$

$$\sigma_d^2 \leftarrow \frac{1}{N*H*W}\sum_{i=1}^{N}\sum_{h=1}^{H}\sum_{w=1}^{W} (x_{i,h,w,d}-\mu_d)^2 \tag{3}$$

$$\hat{x}_{i,h,w,d} \leftarrow \frac{x_{i,h,w,d}-\mu_d}{\sqrt{(\sigma_d^2+\epsilon)}} \tag{4}$$

$$y_{i,h,w,d} \leftarrow \gamma\hat{x}_{i,h,w,d}+\beta \tag{5}$$

where N, H and W represents batch size, height and width.

- "Template/Feature more present *than usual* or not"
- During inference we use stored values for μ_d and σ_d.
- `scale` and `center` params in BatchNormalization layer corresponds to γ and β respectively.

Data augmentation

Idea: apply random transformation to X that does not alter Y.

- Normally you would like result X' to be *plausible*, i.e. *could* have been a sample from the distribution of interest
- Which transformation you may use is application-dependent.
- May also have transformations that change Y as long as we know the effect. E.g. flipping image and label image for semantic segmentation.

Image data

- Horizontal mirroring (issue for objects not left/right symmetric)
- Random crop
- Scale
- Aspect ratio
- Lightning etc.

Text data

- Synonym insertion
- *Back-translation*: translate and translate back with e.g. Google Translate!!!

Reguluarization summary

- Data augmentation - randomness in *input* ==> “increases” training data set
- Dropout - randomness in *activations*
- Batch normalization - randomness in *activations*
- Usually *either* dropout or batch normalization enough
- Weight regularization - penalizes large weights (“non-smooth function”)

Hyperparameters

Hyperparameters to search

From Wikipedia:

```
In machine learning, a hyperparameter is a parameter
whose value is used to control the learning process.
By contrast, the values of other parameters (typically
node weights) are derived via training.
```

Important examples are:

- Learning rate (and learning rate schedule)
- Regularization params: L2, (dropout)
- Model architecture
    - What is the search space?

Search strategies

- Random search rather than grid search
- Logscale when appropriate
- Careful with best values on border
- May refine search

Getting started

Install TensorFlow

Need Python 3.5-3.8. In case of GPU, install prerequisites first.
On Linux/Ubuntu (without virtual environment)

```
pip3 install --upgrade pip
pip3 install --user tensorflow>=2
```

- See https://www.tensorflow.org/install for more.
- API: https://www.tensorflow.org/api_docs/python/
- Tutorials: https://www.tensorflow.org/tutorials

Why TensorFlow / machine learning framework

- Automatic differentiation
- High-level APIs for deep learning (Keras), yet flexible
- Predefined/pretrained models
- Speed - optimized implementation accross devices
  - C++ on CPU
  - CUDA on Nvidia GPUs
  - TPU
  - Embedded devices

Predifined/pretrained models

```python
from matplotlib import pyplot as plt

model = tf.keras.applications.NASNetMobile(weights="imagenet")
image_file = tf.keras.utils.get_file("dog.jpg", "https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcRDg8cZEWA2vuBWvVwGqgilDtKtJKQN-vr17AwwiyOF8C1id81J")
image = plt.imread(image_file)
plt.imshow(image); plt.show()

# resize and preprocess to what model expects
image = tf.expand_dims(tf.image.resize(image, [224, 224]), 0)
image = tf.keras.applications.nasnet.preprocess_input(image)
p = model(image)[0] # 1000-dimensional vector with probabilities
```

```python
# Show labels and probabilty for top5 predictions
sorted_indices = tf.argsort(p, direction="DESCENDING")
labels_path = tf.keras.utils.get_file('ImageNetLabels.txt','https://storage.googleapis.com/download.tensorflow.org/data/ImageNetLabels.txt')
imagenet_labels = np.array(open(labels_path).read().splitlines())
for idx in sorted_indices[:5]:
  print("%25s: %g" % (imagenet_labels[idx], p[idx]))
```

- Also unofficial/random models from community
- Use as part of larger system and/or finetuning.