UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

3253 Machine Learning

Module 10: Introduction to Deep Learning and Deep Neural Networks

Course Plan

Module Titles
Module 1 – Introduction to Machine Learning
Module 2 – End to End Machine Learning Project
Module 3 – Classification
Module 4 – Clustering and Unsupervised Learning
Module 5 – Training Models and Feature Selection
Module 6 – Support Vector Machines
Module 7 – Decision Trees and Ensemble Learning
Module 8 – Dimensionality Reduction
Module 9 – Introduction to TensorFlow
Module 10 – **Current Focus: Intro to Deep Learning & Deep Neural Networks**
Module 11 – Distributing TensorFlow, CNNs and RNNs
Module 12 – Final Assignment and Presentations (no content)

Learning Outcomes for this Module

- Understand the structure of Neural Networks
- Choose hyperparameter settings for learning a net
- Discuss training Neural Networks
  - Gradient Problems
  - Optimizers
  - Regularization

Topics for this Module

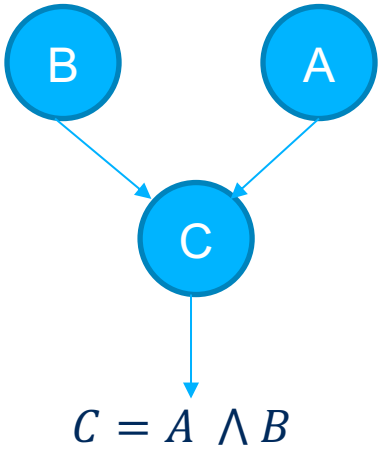
- **10.1** Artificial neural networks
- **10.2** Training neural networks
- **10.3** Deep networks
- **10.4** Deep networks using TensorFlow
- **10.5** Training deep networks
- **10.6** Improving on ReLU
- **10.7** Normalization
- **10.8** Gradient Clipping
- **10.9** Architecture
- **10.10** Optimization
- **10.11** Regularization
- **10.12** Pre-training
- **10.13** Learning Rates
- **10.14** Resources and Wrap-up

Module 10 – Section 1

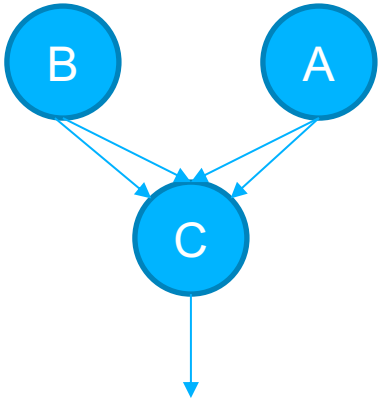
Artificial Neural Networks

Neuron

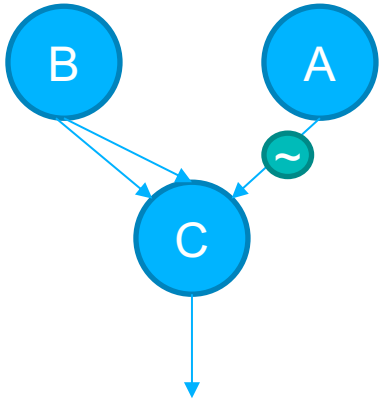
- Simplified model, initially introduced by McCulloch and Pitts (1943):
  - Binary artificial neuron
  - Two inputs, one output
- Can be combined to make complicated expressions
- Below: output is active when at least two inputs are active



$C = A \wedge B$



$C = A \vee B$

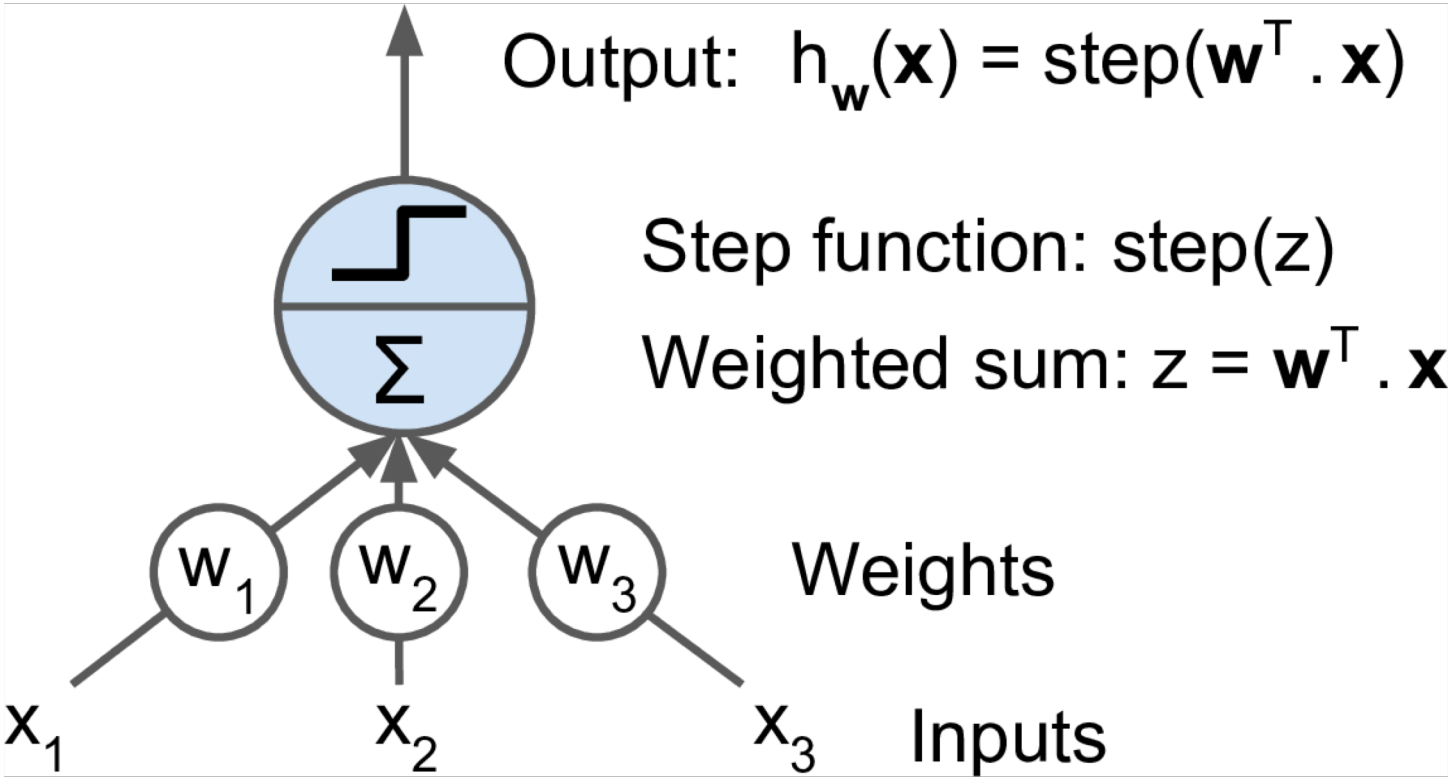


$C = B \wedge \sim A$

Perceptrons

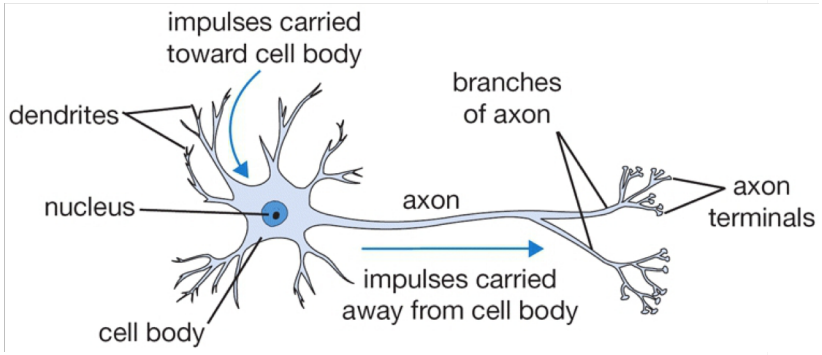
- Rosenblatt (1957): Now inputs can be floats
- Each input has an associated weight
- Weighted sum is put through a step (Heaviside) function
- $\text{heaviside(z)} = \begin{cases} 0 \; if \; z < 0 \\ 1 \; if \; z \geq 0 \end{cases}$
- $z = w_1 x_1 + w_2 x_2 + \; ... + w_n x_n$
- $h_w(z) = heaviside(z)$

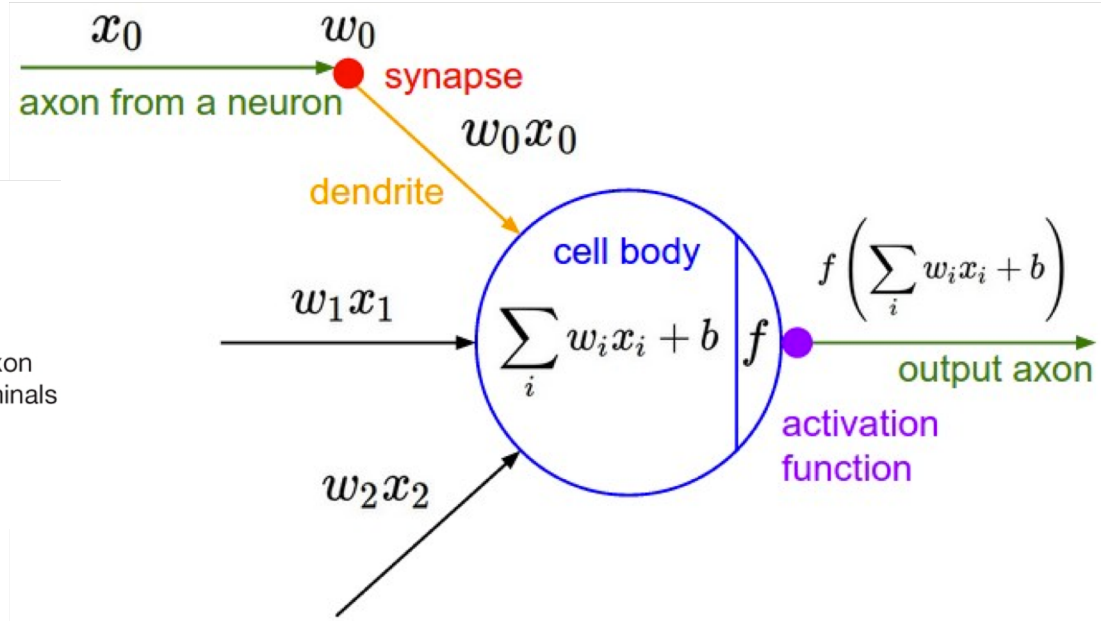
Perceptrons (cont'd)



Neurons

- Neural networks define functions of the inputs (hidden features), computed by neurons
- Artificial neurons are called units





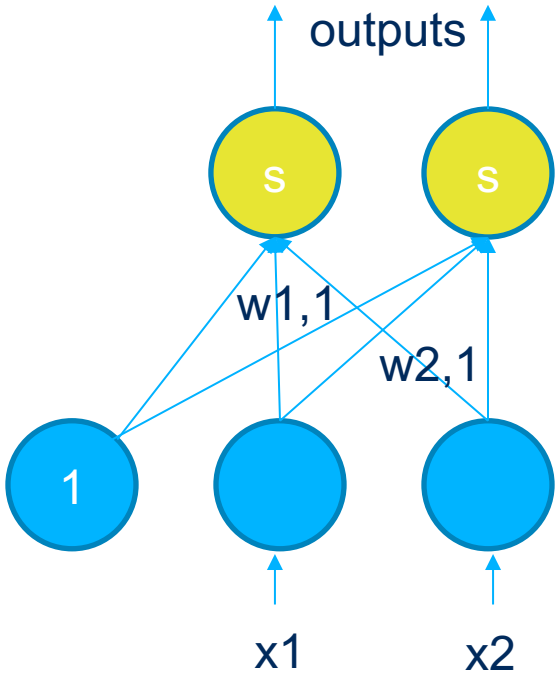
http://cs231n.github.io/neural-networks-1/

Perceptrons

- Each input is associated with a weight
- Can be used for simple binary classification
- Computes linear combination of inputs and if result exceeds a threshold, outputs a positive class (otherwise, outputs a negative class)
- Feed input neurons along with bias neuron

Perceptrons (cont'd)

- Edges are weights
- Green "s" circles are step functions
- Lower left circle is bias



UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 10 – Section 2

Training Neural Networks

Training a Perceptron

- $w_{i,j} = w_{i,j} + n\left(\hat{y}_j - y_j\right)x_i$ where
- $w_{i,j}$ connection weight between i-th input and j-th output
- x_i i-th input
- $\hat{y}_j$ output of the of the j-th neuron
- y_j target value

Training a Perceptron (cont'd)

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)]  # petal length, petal width
y = (iris.target == 0).astype(np.int)  # Iris Setosa?

per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

Hyperparameters

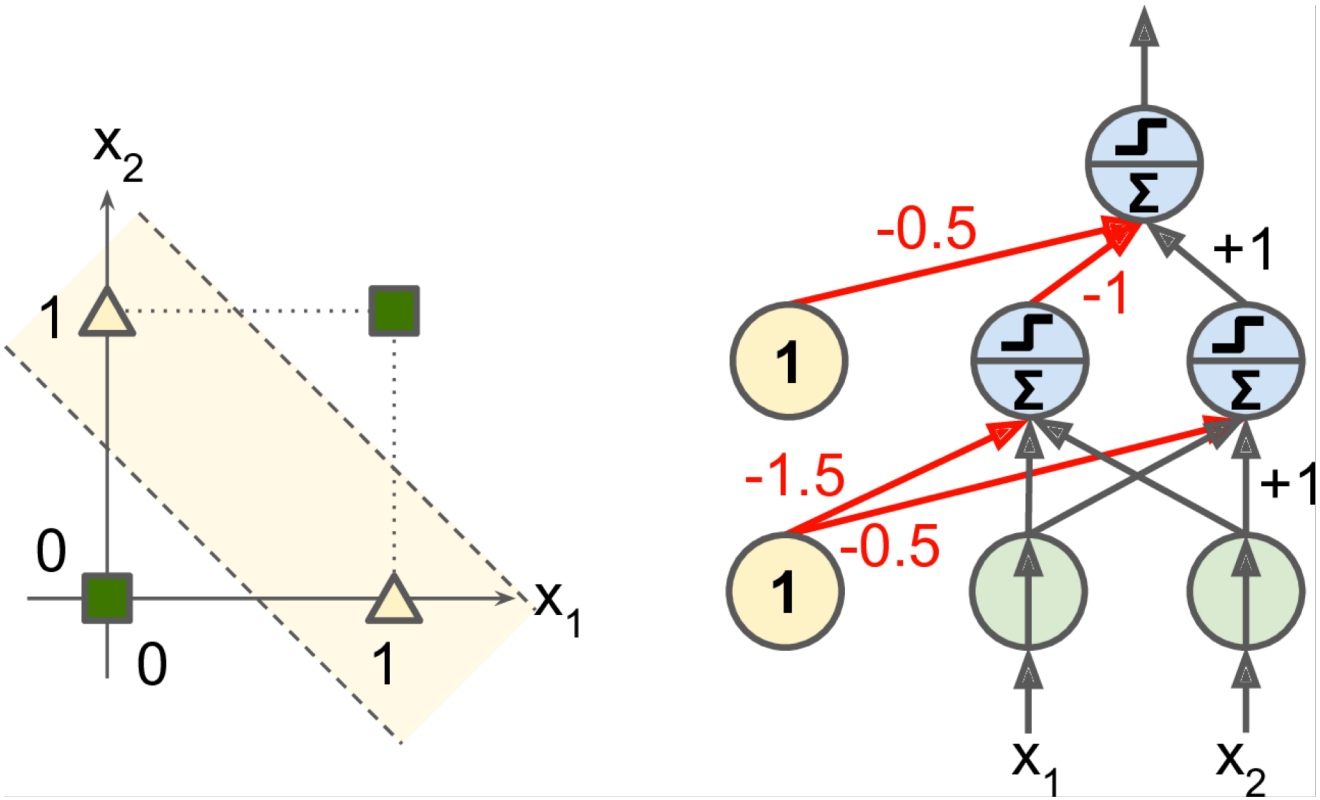
- In simple MLP:
  - Number of layers,
  - Number of neurons per layer,
  - Type of activation function,
  - Weight initialization, etc. are all variable

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 10 – Section 3

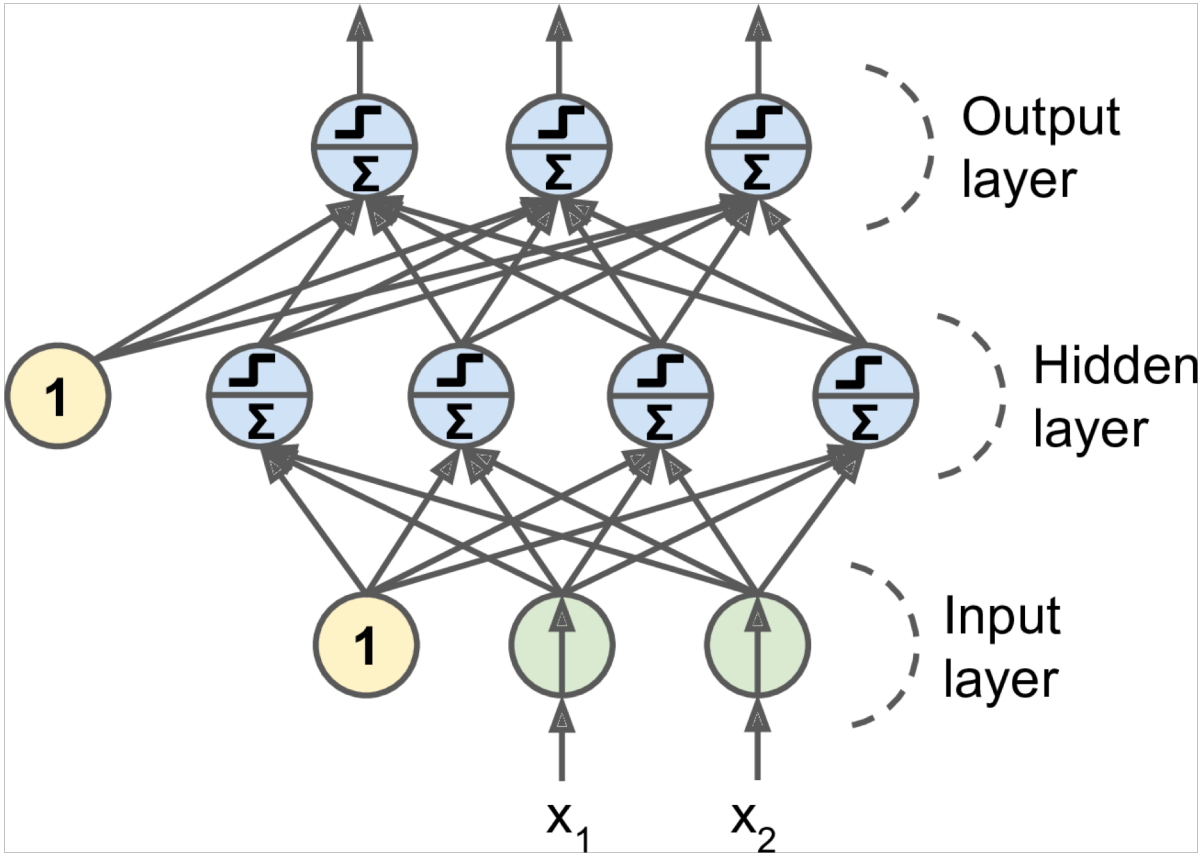
Deep Networks

Multiple Layers



One layer alone cannot solve the problem (Non-linear separable)

Multiple Layers (cont'd)



Number of Layers

- For many problems, can just start with one hidden layer and will see acceptable results
- Deeper networks can model a complex function with exponentially fewer neurons: have higher *parameter efficiency*
- With deep convolutional networks (specific type of neural network), first layers model low-level structures: line segments for example
- Subsequent layers will model intermediate level structures (various shapes)
- Deepest layers will model high level structure (eg: face)

Number of Layers (cont'd)

- Standard strategy is to continue to add layers until you experience overfitting
- Yoshua Bengio: "*Very simple. Just keep adding layers until the test error does not improve anymore.*" [1]
- Geoffrey Hinton: Add more layers until you get overfitting, then add dropout.

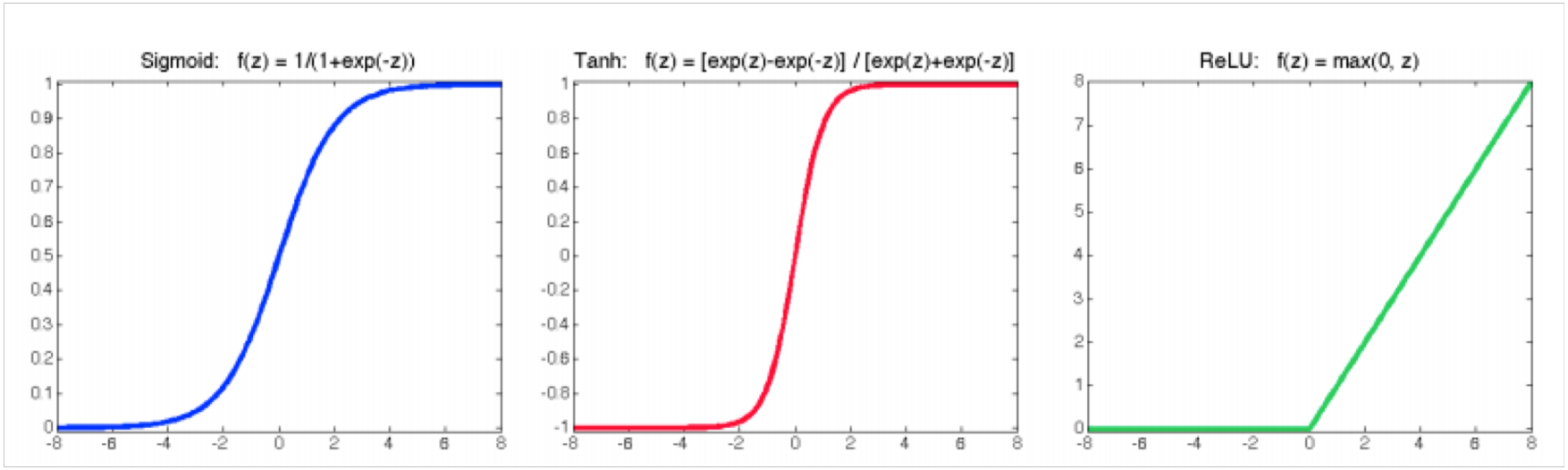
[1] https://www.quora.com/Artificial-Neural-Networks/Artificial-Neural-Networks-How-can-I-estimate-the-number-of-neurons-and-layers/answer/Yoshua-Bengio?share=7b58dc3b&srid=hqJd

Non-saturating Activation Functions

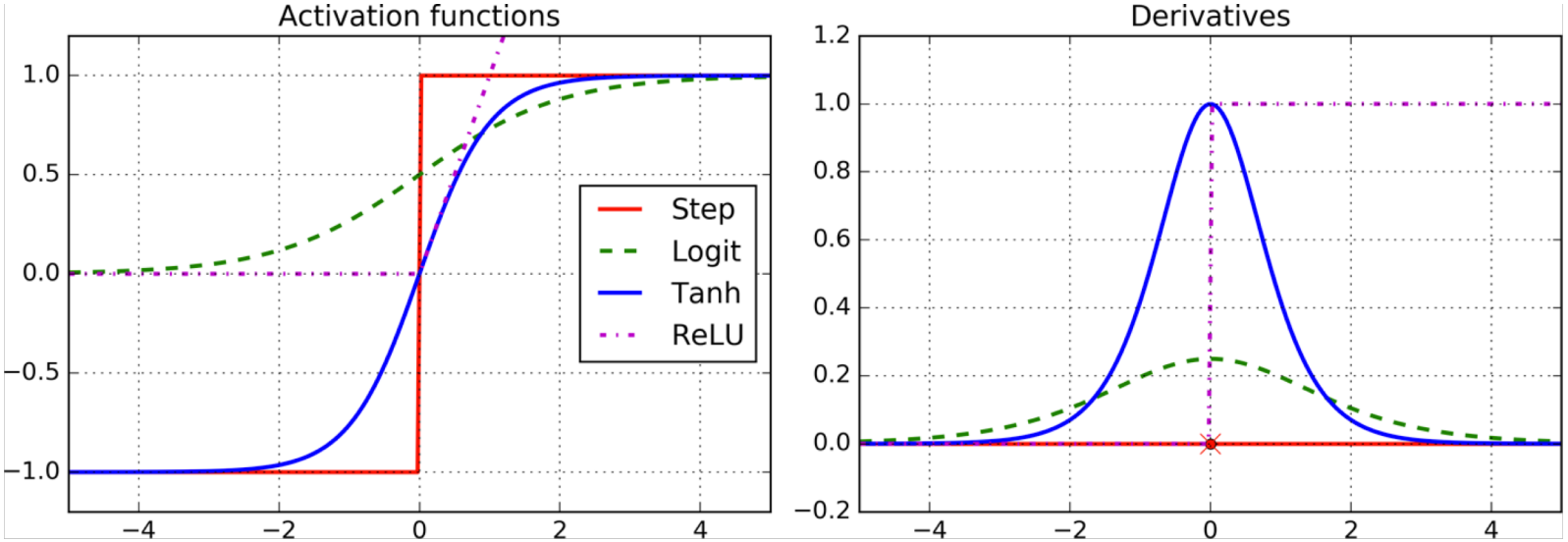
- ReLU does not saturate for positive values
- $relu(z) = \max(0, z)$
- Has problem of *dying ReLUs*: during training, some neurons begin only outputting zero
  - If weighted sum of neuron's inputs is negative, this will happen
  - Gradient of ReLU function is zero when input is negative
- $lrelu_{\alpha}(z) = \max(\alpha z, z)$
- α is slope of function when z<0, determines how much it "leaks"
- Ensures that lrelu will not die

Activation Functions

- Most commonly used activation functions:
  - Sigmoid $\sigma(z) = \frac{1}{1+\exp(-z)}$
  - Tanh $\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$
  - ReLU (Rectifier Linear Unit) $\mathrm{ReLU}(z) = \max(0, z)$

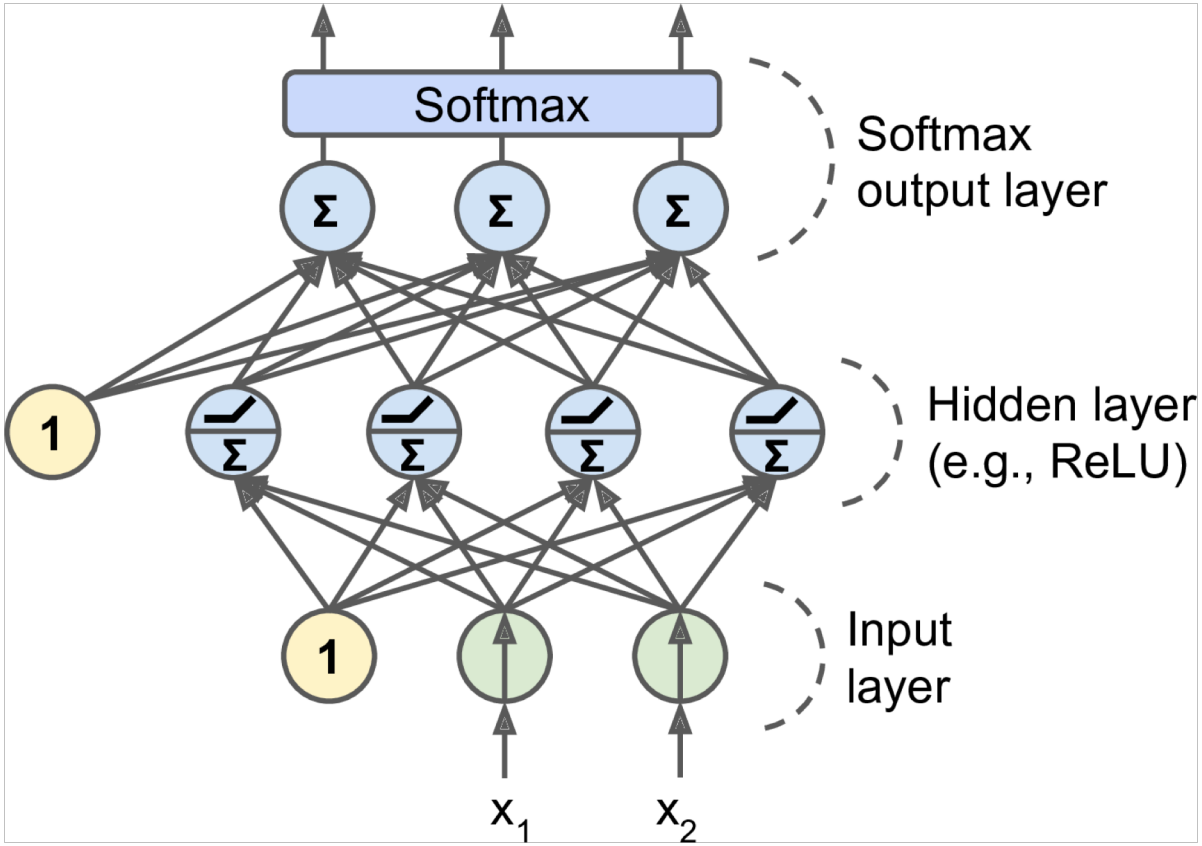


Activation Functions (cont'd)



Activation functions and their derivatives

MLP for Classification



Back-Propagation

- For each training instance the back-propagation algorithm:
  - first makes a prediction (forward pass),
  - measures the error,
  - then goes through each layer in reverse to measure the error contribution from each connection (reverse pass),
  - and finally slightly tweaks the connection weights to reduce the error (Gradient Descent step).

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 10 – Section 4

Deep Networks using TensorFlow

MLP with TF

- The simplest way to train an MLP with TensorFlow is to use the high-level API TF.Learn.
  - The DNNClassifier class makes it easy to train a deep neural network with any number of hidden layers, and a softmax output layer to output estimated class probabilities.

```
import tensorflow as tf

feature_cols = tf.contrib.learn.infer_real_valued_columns_from_input(X_train)
dnn_clf = tf.contrib.learn.DNNClassifier(hidden_units=[300,100], n_classes=10,
                                         feature_columns=feature_cols)
dnn_clf = tf.contrib.learn.SKCompat(dnn_clf)  # if TensorFlow >= 1.1
dnn_clf.fit(X_train, y_train, batch_size=50, steps=40000)
```

This would produce 98.2% accuracy on the test set!

MLP with Plain TF

- Specify parameters:

```
import tensorflow as tf

n_inputs = 28*28  # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10
```

- Use placeholder nodes to represent the training data and targets. The shape of X is only partially defined.

```
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

neuron_layer

```
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 2 / np.sqrt(n_inputs + n_neurons)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="kernel")
        b = tf.Variable(tf.zeros([n_neurons]), name="bias")
        Z = tf.matmul(X, W) + b
        if activation is not None:
            return activation(Z)
        else:
            return Z
```

Build the Network

```
with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, name="hidden1",
                           activation=tf.nn.relu)
    hidden2 = neuron_layer(hidden1, n_hidden2, name="hidden2",
                           activation=tf.nn.relu)
    logits = neuron_layer(hidden2, n_outputs, name="outputs")
```

Alternatively

```
with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1",
                              activation=tf.nn.relu)
    hidden2 = tf.layers.dense(hidden1, n_hidden2, name="hidden2",
                              activation=tf.nn.relu)
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs")
```

Cost Function + Training

- Define the cost function

```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                              logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")
```

- Define optimizer

```
learning_rate = 0.01

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)
```

Model Evaluation

- Specify how to evaluate the model:

```
with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

- Create a node to initialize all variables, and we will also create a Saver to save our trained model parameters to disk:

```
init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 10 – Section 5

Training Deep Networks

Train the Model

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/")
```

```
n_epochs = 40
batch_size = 50
```

```
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_val = accuracy.eval(feed_dict={X: mnist.validation.images,
                                           y: mnist.validation.labels})
        print(epoch, "Train accuracy:", acc_train, "Val accuracy:", acc_val)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

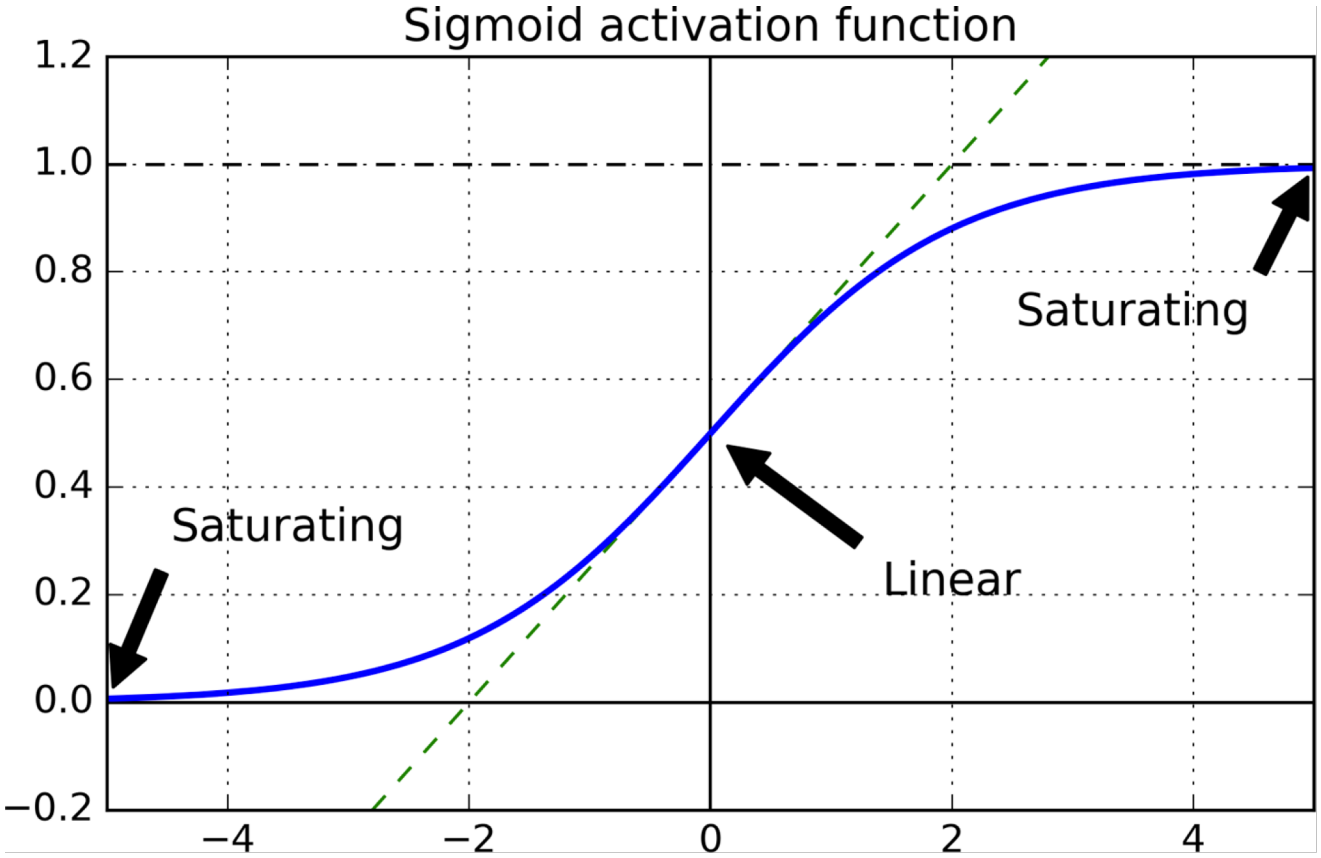
DNNs Challenges

- V*anishing gradients* problem (or the related *exploding gradients* problem) that affects deep neural networks and makes lower layers very hard to train.
- For large networks, training can be extremely slow.
- A model with millions of parameters would severely risk overfitting the training set.

Vanishing/Exploding Gradients

- Gradients often get smaller and smaller as the algorithm progresses down to the lower layers.
  - The Gradient Descent update leaves the lower layer connection weights virtually unchanged, and training never converges to a good solution.
  - In some cases: The gradients can grow bigger and bigger, so many layers get large weight updates and the algorithm diverges
  - More generally, deep neural networks suffer from unstable gradients; different layers may learn at widely different speeds.

Vanishing/Exploding Gradients (cont'd)



Vanishing/Exploding Gradients (cont'd)

- We don't want the signal to die out, nor do we want it to explode and saturate.
  - For the signal to flow properly, the authors argue that we need the variance of the outputs of each layer to be equal to the variance of its inputs [Glorot and Bengio]
  - Solution: Better initialization (Xavier)

Normal distribution with mean 0 and standard deviation $\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

Or a uniform distribution between -r and +r, with $r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$

Vanishing/Exploding Gradients (cont'd)

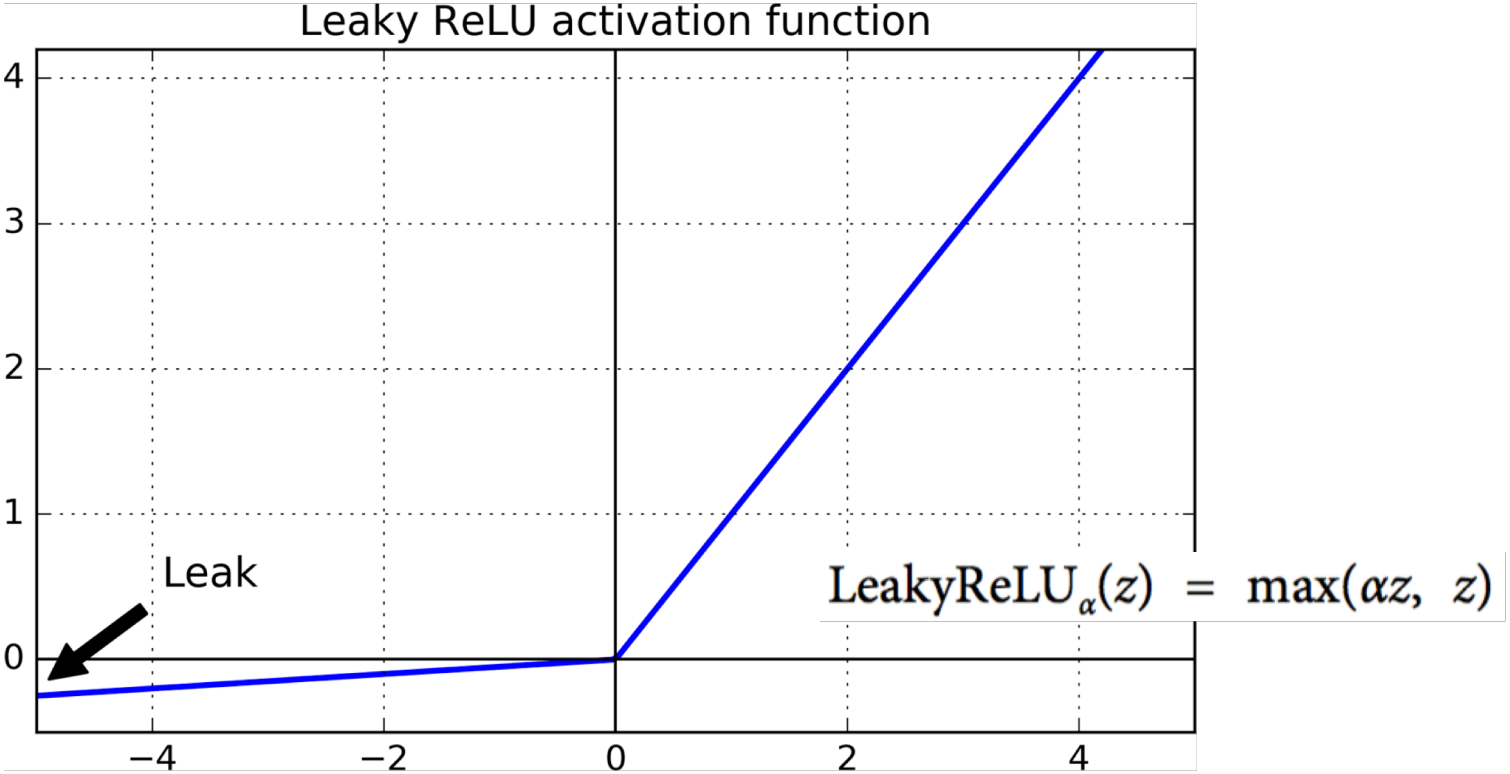
Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
ReLU (and its variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$

```
he_init = tf.contrib.layers.variance_scaling_initializer()
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
                          kernel_initializer=he_init, name="hidden1")
```

Module 10 – Section 6

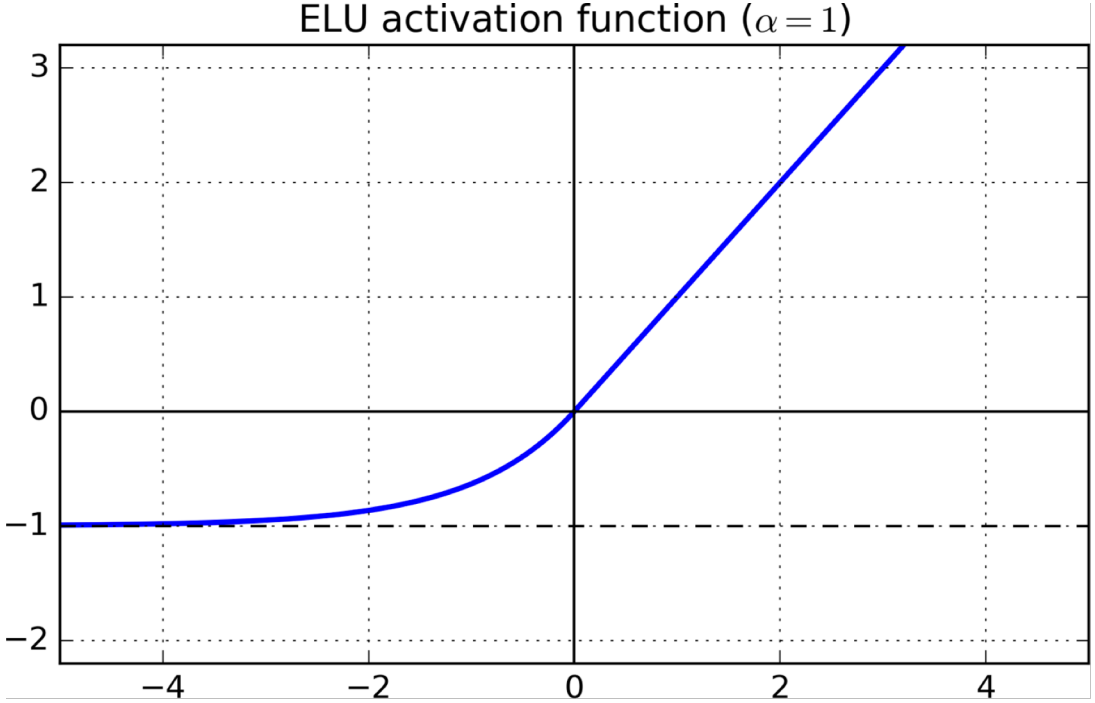
Improving on ReLU

Leaky ReLU



The ReLU activation function is not perfect. It suffers from a problem known as the *dying ReLUs*: during training, some neurons effectively die, meaning they stop outputting anything other than 0.

Exponential Linear Unit



$$\mathrm{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

Djork-Arné Clevert et al. proposed a new activation function called the *exponential linear unit* (ELU) that outperformed all the ReLU variants in their experiments: training time was reduced and the neural network performed better on the test set.

The main drawback of the ELU activation function is that it is slower to compute but during training this is compensated by the faster convergence rate.

```
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.elu, name="hidden1")
```

Which Activation Functions to Choose

- In general
  - ELU > leaky ReLU (and its variants) > ReLU > tanh > logistic.
  - If you care a lot about runtime performance, then you may prefer leaky ReLUs over ELUs.
  - If you don't want to tweak yet another hyperparameter, you may just use the default α values (0.01 for the leaky ReLU, and 1 for ELU).
  - If you have spare time and computing power, you can use cross-validation to evaluate other activation functions, in particular RReLU if your network is overfitting, or PReLU if you have a huge training set.

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 10 – Section 7

Normalization

Batch Normalization (BN)

- BN consists of adding an operation in the model just before the activation function of each layer
  - Zero-centering and normalizing the inputs, then scaling and shifting the result using two new parameters per layer (one for scaling, the other for shifting).
  - This operation lets the model learn the optimal scale and mean of the inputs for each layer.
  - Used to fight vanishing/exploding gradients.
- *Internal covariate shift*: distribution of each layer's inputs changes during training
- $\mu_B = \frac{1}{m}\sum_{i=1}^{m_B} x^{(i)}$, $\sigma_B{}^2 = \frac{1}{m}\sum_{i=1}^{m_B}\left(x^{(i)} - \mu_B\right)^2$
- $\hat{x}^{(i)} = \frac{x^{(i)} - \mu_B}{\sqrt{\sigma_B{}^2 + \epsilon}}$, $z^{(i)} = \gamma\ \hat{x}^{(i)} + \beta$
- At test time there's no empirical mean / std dev so mean from training is used
- Prior to batch norm, gradient clipping was popular to ensure that gradients never exceed a threshold (use tf.clip_by_value)

Batch Normalization (cont'd)

```
import tensorflow as tf

n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")

training = tf.placeholder_with_default(False, shape=(), name='training')

hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1")
bn1 = tf.layers.batch_normalization(hidden1, training=training, momentum=0.9)
bn1_act = tf.nn.elu(bn1)
hidden2 = tf.layers.dense(bn1_act, n_hidden2, name="hidden2")
bn2 = tf.layers.batch_normalization(hidden2, training=training, momentum=0.9)
bn2_act = tf.nn.elu(bn2)
logits_before_bn = tf.layers.dense(bn2_act, n_outputs, name="outputs")
logits = tf.layers.batch_normalization(logits_before_bn, training=training,
                                       momentum=0.9)
```

Batch Normalization (cont'd)

```
from functools import partial

my_batch_norm_layer = partial(tf.layers.batch_normalization,
                              training=training, momentum=0.9)

hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1")
bn1 = my_batch_norm_layer(hidden1)
bn1_act = tf.nn.elu(bn1)
hidden2 = tf.layers.dense(bn1_act, n_hidden2, name="hidden2")
bn2 = my_batch_norm_layer(hidden2)
bn2_act = tf.nn.elu(bn2)
logits_before_bn = tf.layers.dense(bn2_act, n_outputs, name="outputs")
logits = my_batch_norm_layer(logits_before_bn)
```

- To avoid code repetition, you can use the partial() function from the functools module (part of Python's standard library). It creates a thin wrapper around a function and allows you to define default values for some parameters.

Batch Normalization Execution

- During training, you need to set the training placeholder to True.
- The batch_normalization() function creates a few operations that must be evaluated at each step during training in order to update the moving averages.
- These operations are automatically added to the UPDATE_OPS collection.

```
extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run([training_op, extra_update_ops],
                     feed_dict={training: True, X: X_batch, y: y_batch})
        accuracy_val = accuracy.eval(feed_dict={X: mnist.test.images,
                                                y: mnist.test.labels})
        print(epoch, "Test accuracy:", accuracy_val)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 10 – Section 8

Gradient Clipping

Gradient Clipping

- A popular technique to lessen the exploding gradients problem is to simply clip the gradients during backpropagation so that they never exceed some threshold

```
threshold = 1.0
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(loss)
capped_gvs = [(tf.clip_by_value(grad, -threshold, threshold), var)
              for grad, var in grads_and_vars]
training_op = optimizer.apply_gradients(capped_gvs)
```

- In general, batch normalization is preferred

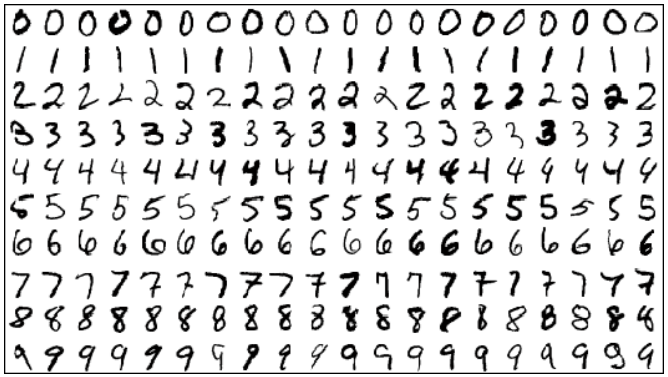
UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 10 – Section 9

Architecture

Number of Neurons per Layer

- Number of input and output neurons dictated by task: MNIST images are 28x28x1 (1 is the number of channels) so inputs= 784
- Ten possible outputs (0-9), 10 outputs



Number of Neurons per Hidden Layer

- Common strategy is to use a funnel approach
- Fewer and fewer neurons at each layer
- Idea is that many low-level features can “coalesce” into fewer high-level features
- Like number of layers, can try increasing the number of neurons until overfitting happens
- Will get much more “bang for your buck” by adding more layers, however

Module 10 – Section 10

Optimization

Optimizers

- Training a large DNN can be slow
- One important speed-up can come from using a faster optimizer than the regular Gradient Descent optimizer.
- Most popular:
  - Momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, and Adam optimization.

Momentum Optimization

- Imagine a bowling ball rolling down a gentle slope on a smooth surface:
  - it will start out slowly, but it will quickly pick up momentum until it eventually reaches terminal velocity
- Momentum optimization cares a great deal about what previous gradients were: at each iteration, it subtracts the local gradient from the *momentum vector* **m** (multiplied by the learning rate η), and it updates the weights by adding this momentum vector.

- The gradient is used as an acceleration, not as a speed.

1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta + \mathbf{m}$

Momentum Optimization (cont'd)

- The optimizer may overshoot a bit, then come back, overshoot again, and oscillate like this many times before stabilizing at the minimum.
- This is one of the reasons why it is good to have a bit of friction in the system: it gets rid of these oscillations and thus speeds up convergence.

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9)
```

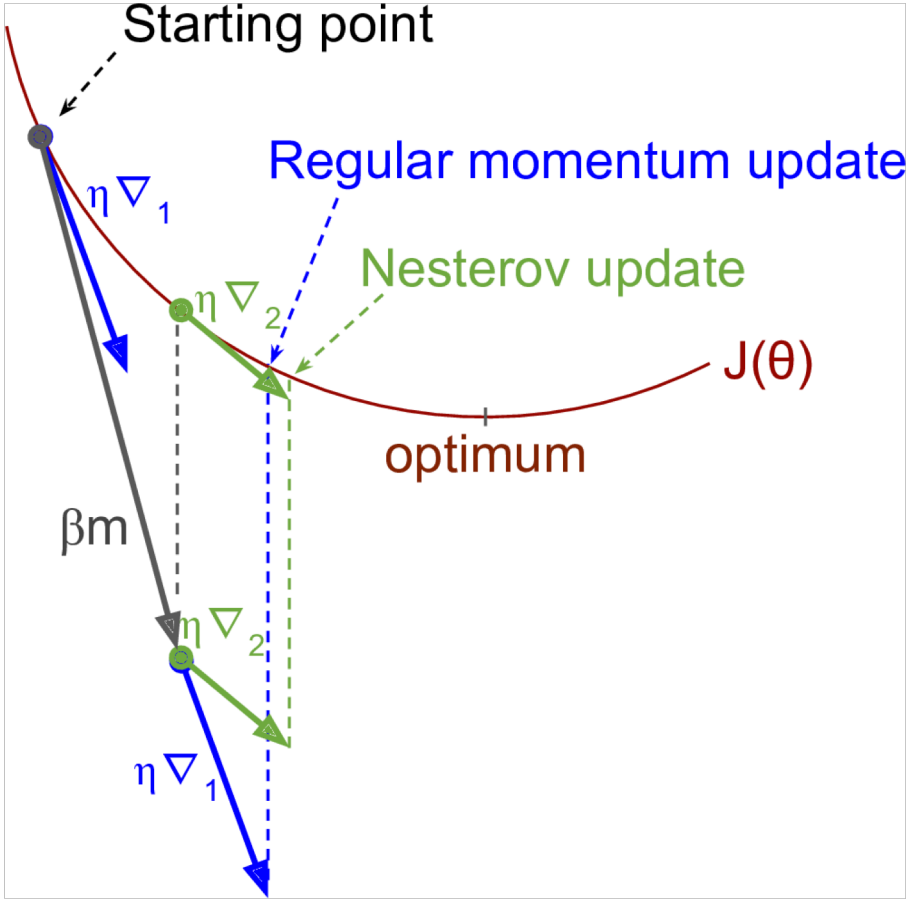
- Another hyperparameter to tune!

Nesterov Accelerated Gradient

Nesterov Accelerated Gradient (NAG):

- Measure the gradient of the cost function not at the local position but slightly ahead in the direction of the momentum.
- The only difference from vanilla Momentum optimization is that the gradient is measured at $\theta + \beta\mathbf{m}$ rather than at θ.

1. $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta \nabla_\theta J(\theta + \beta\mathbf{m})$
2. $\theta \leftarrow \theta + \mathbf{m}$




```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9, use_nesterov=True)
```

AdaGrad

- Gradient Descent starts by quickly going down the steepest slope, then slowly goes down the bottom of the valley.
  - Algorithm corrects its direction to point a bit more toward the global optimum.
- AdaGrad scales down the gradient vector along the steepest dimensions

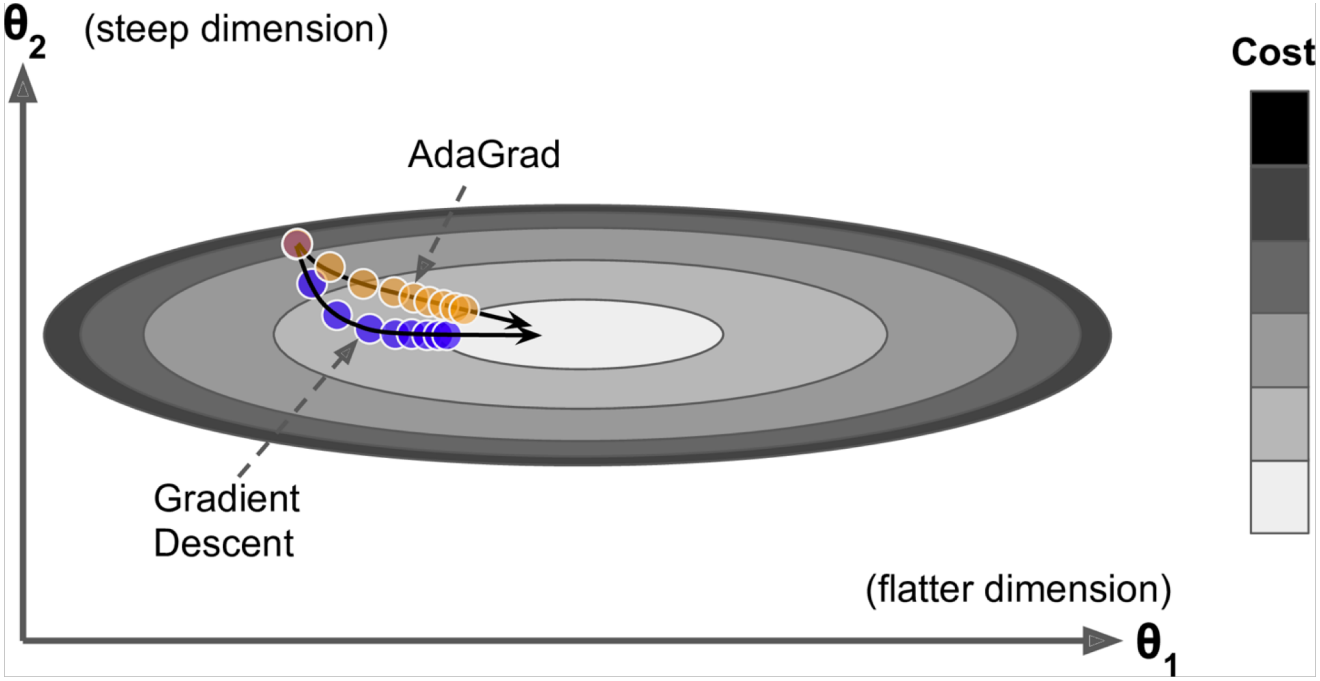
$$1. \quad \mathbf{s} \leftarrow \mathbf{s} + \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

$$2. \quad \theta \leftarrow \theta - \eta \, \nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$$

- AdaGrad decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an *adaptive learning rate*. It helps point the resulting updates more directly toward the global optimum

AdaGrad (cont'd)

- AdaGrad often performs well for simple quadratic problems, but unfortunately it often stops too early when training neural networks.



RMSProp

- *RMSProp* fixes AdaGrad by accumulating only the gradients from the most recent iterations
  - It does so by using exponential decay in the first step

$$1. \quad \mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$$

$$2. \quad \theta \leftarrow \theta - \eta \, \nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$$

```
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate,
                                      momentum=0.9, decay=0.9, epsilon=1e-10)
```

Adam Optimization

ADAM (adaptive moment estimation)

- Combines the ideas of Momentum optimization and RMSProp: just like Momentum optimization it keeps track of an exponentially decaying average of past gradients, and just like RMSProp it keeps track of an exponentially decaying average of past squared gradients

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_\theta J(\theta)$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
3. $\mathbf{m} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^t}$
4. $\mathbf{s} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^t}$
5. $\theta \leftarrow \theta + \eta \, \mathbf{m} \oslash \sqrt{\mathbf{s} + \epsilon}$

Adam is an adaptive learning rate algorithm (like AdaGrad and RMSProp), it requires less tuning of the learning rate hyperparameter η. You can often use the default value $\eta = 0.001$

```
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

Warning

- One of the most common approaches is to use Adam optimization, because it was generally considered faster and better than other methods.
- However, a 2017 paper by Ashia C. Wilson et al. (https://arxiv.org/abs/1705.08292) showed that adaptive optimization methods (i.e., AdaGrad, RMSProp and Adam optimization) can lead to solutions that generalize poorly on some datasets.
- So you may want to stick to Momentum optimization or Nesterov Accelerated Gradient for now, until researchers have a better understanding of this issue.

Module 10 – Section 11

Regularization

Regularization

- Can mitigate overfitting to a degree with regularization during training
- *Early stopping*: stop training when performance on validation set starts to drop
- *L1/L2 regularization*: add a term to the loss that penalizes the L1 or L2 norm of the weights

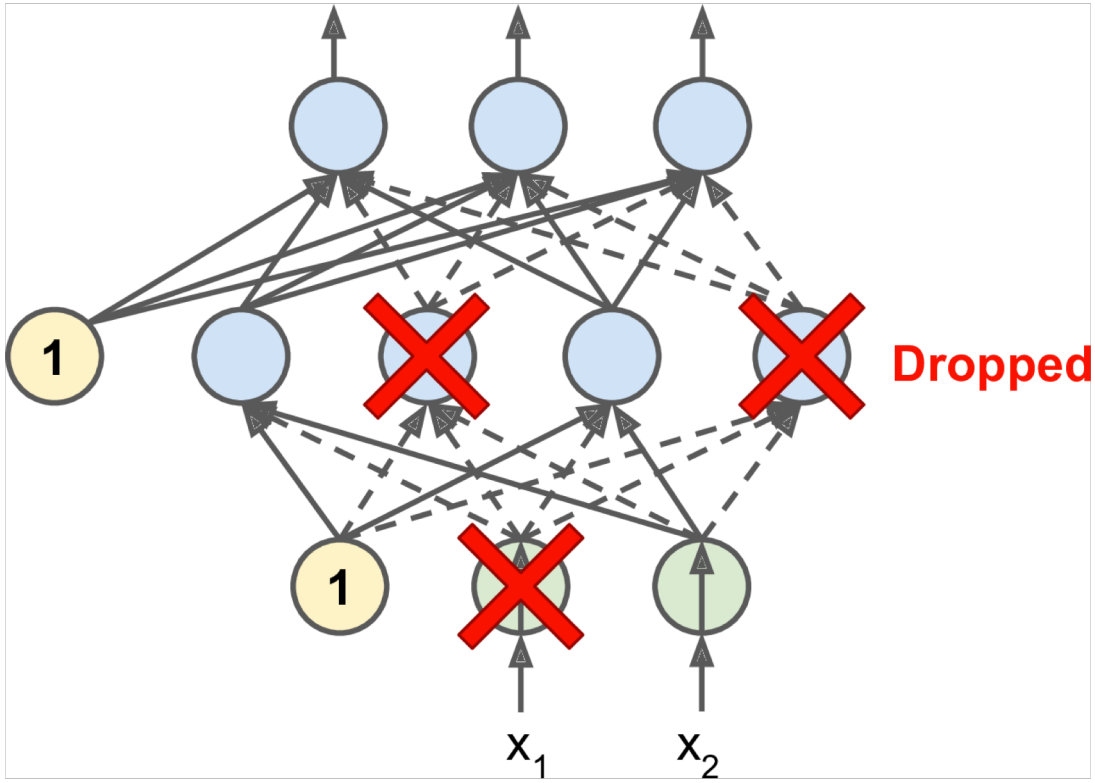
Regularization (cont'd)

```
[...] # construct the neural network
W1 = tf.get_default_graph().get_tensor_by_name("hidden1/kernel:0")
W2 = tf.get_default_graph().get_tensor_by_name("outputs/kernel:0")

scale = 0.001 # l1 regularization hyperparameter

with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                              logits=logits)
    base_loss = tf.reduce_mean(xentropy, name="avg_xentropy")
    reg_losses = tf.reduce_sum(tf.abs(W1)) + tf.reduce_sum(tf.abs(W2))
    loss = tf.add(base_loss, scale * reg_losses, name="loss")
```

Many functions that create variables (such as get_variable() or tf.layers.dense()) accept a regularizer argument for each created variable (e.g., kernel_regularizer)

```
my_dense_layer = partial(
    tf.layers.dense, activation=tf.nn.relu,
    kernel_regularizer=tf.contrib.layers.l1_regularizer(scale))

with tf.name_scope("dnn"):
    hidden1 = my_dense_layer(X, n_hidden1, name="hidden1")
    hidden2 = my_dense_layer(hidden1, n_hidden2, name="hidden2")
    logits = my_dense_layer(hidden2, n_outputs, activation=None,
                            name="outputs")
```

Dropout



Dropout (cont'd)

It is a fairly simple algorithm:

- at every training step, every neuron (including the input neurons but excluding the output neurons) has a probability *p* of being temporarily "dropped out," meaning it will be entirely ignored during this training step, but it may be active during the next step
- The hyperparameter *p* is called the *dropout rate*, and it is typically set to 50%. After training, neurons don't get dropped anymore.
- Multiply each input connection weight by the *keep probability* (1 – *p*) after training.

```
[...]
training = tf.placeholder_with_default(False, shape=(), name='training')

dropout_rate = 0.5  # == 1 - keep_prob
X_drop = tf.layers.dropout(X, dropout_rate, training=training)

with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X_drop, n_hidden1, activation=tf.nn.relu,
                              name="hidden1")
    hidden1_drop = tf.layers.dropout(hidden1, dropout_rate, training=training)
    hidden2 = tf.layers.dense(hidden1_drop, n_hidden2, activation=tf.nn.relu,
                              name="hidden2")
    hidden2_drop = tf.layers.dropout(hidden2, dropout_rate, training=training)
    logits = tf.layers.dense(hidden2_drop, n_outputs, name="outputs")
```

Module 10 – Section 12

Pre-Training

Pretraining

- Often a good idea to use learned weights trained from a different task
- Say network A was originally trained to classify cats and dogs, new network B will be used to classify trucks and cars
- Remove final logistic layer(s)
- Can take well trained weights of network A and adapt them to new task:
  - Possibly delete the final layer (or layers), lock all remaining weights
  - Add new final layers for classification and only train the new weights
  - Network B will learn new weighting of existing features to accurately describe cars and trucks
- Especially useful when training set is small

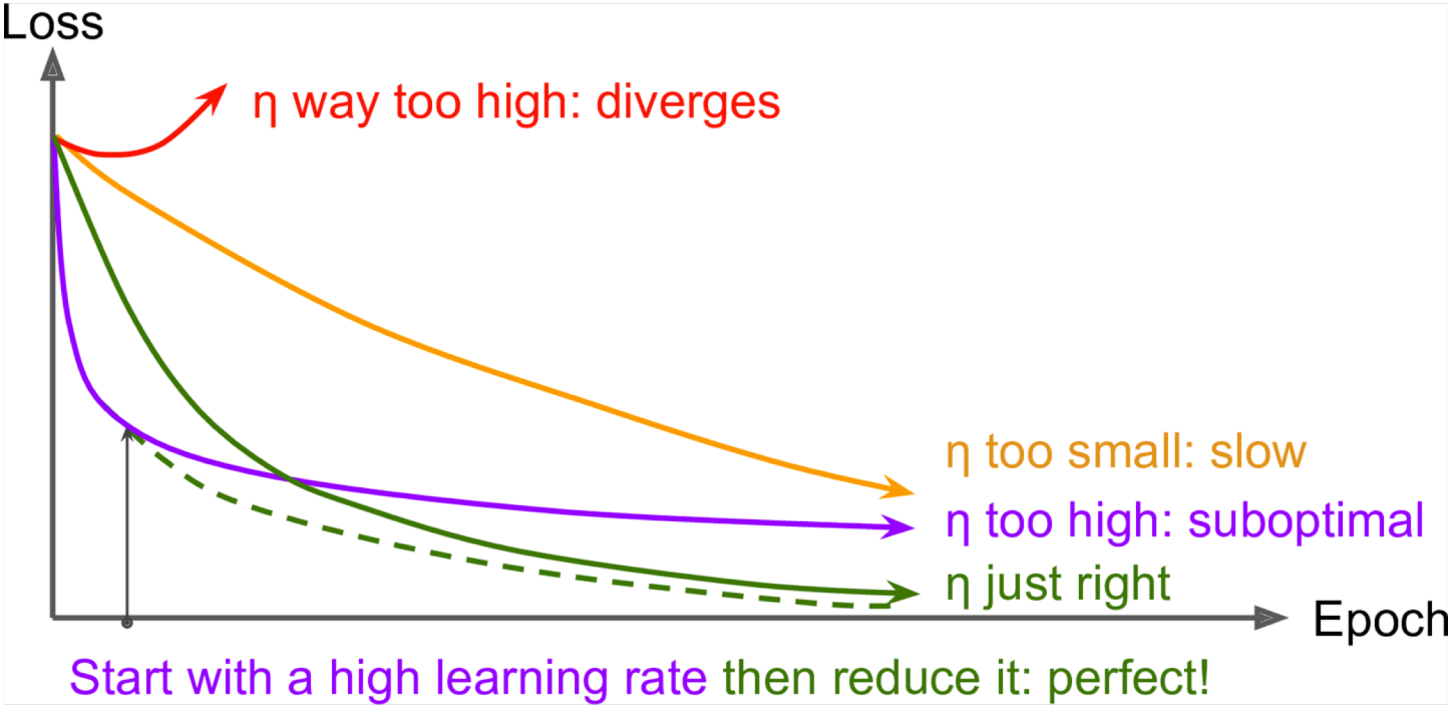
Module 10 – Section 13

Learning Rates

Finding a Good Learning Rate

- Usually the most important hyperparameter to tune w.r.t. convergence
- Setting learning rate (LR) appropriately is generally a process of trial and error
- Too high and training will actually diverge; too low and it will not converge to the optimum in a reasonable amount of time
- Slightly too high of a LR will result in fast progress (losses will drop quickly) but will often oscillate around the optimum, never settling down (though adaptive optimization like Adam, RMSProp, etc will help with this)
- Ideal learning rate will learn quickly (i.e. losses drop quickly) and converge to a good solution

Learning Rate Scheduling



Learning Rate Scheduling

- Predetermined piecewise constant scheduling - start with learning rate n_0 after m epochs switch to n_1, where all three values are predetermined
- Performance scheduling – measure validation error every N steps and drop learning rate as validation error flattens
- Exponential scheduling – Set learning rate to function of iteration number, for example

  $\mathrm{n}(t) = n_0 10^{-t/r}$
- Power scheduling
  - $n(t) = n_0 \left(1 + \frac{t}{r}\right)^{-c}$

Summary

Initialization	He initialization
Activation function	ELU
Normalization	Batch Normalization
Regularization	Dropout
Optimizer	Nesterov Accelerated Gradient
Learning rate schedule	None

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 10 – Section 14

Resources and Wrap-up

Resources

- Hands-On Machine Learning with Scikit-Learn and Tensorflow:
  - Chapter 10, 11

Assessment

- See Jupyter Notebook

Next Class

- Distributing Neural Networks
- Other architectures

Follow us on social

Join the conversation with us online:

facebook.com/uoftscs

@uoftscs

linkedin.com/company/university-of-toronto-school-of-continuing-studies

@uoftscs

Any questions?

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Thank You

Thank you for choosing the University of Toronto School of Continuing Studies