UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

SCS 3253 Analytics Techniques and Machine Learning

Module 9: Introduction to Neural Networks & Deep Learning

Course Plan

Module Titles
Module 1 – Current Focus: 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
Current Focus: Module 9 – Introduction to Neural Networks & Deep Learning
Module 10 – Introduction to TensorFlow
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 training
- Dive into training details for Neural Networks
  - Gradient Problems
  - Optimizers
  - Regularization

Topics in this Module

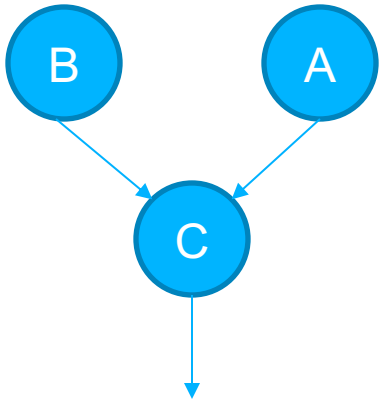
- **9.1** Artificial Neural Networks (ANNs)
- **9.2** Training ANNs
- **9.3** Deep Neural Networks (DNNs)
- **9.4** Training DNNs
- **9.5** Optimizers
- **9.6** Training Challenges
- **9.7** Training Strategies
- **9.8** Wrap-up

Module 9 - Section 1

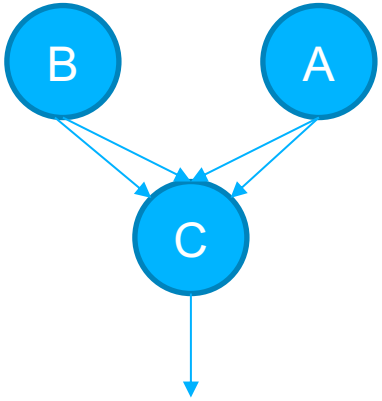
Artificial Neural Networks (ANNs)

Neuron

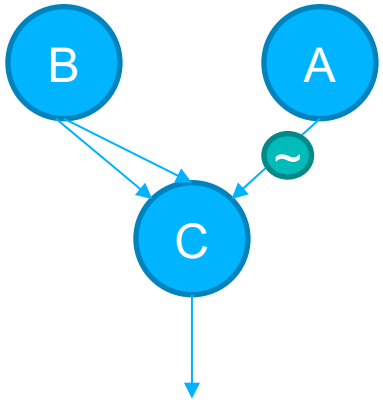
- Simplified model, initially introduced by McCulloch and Pitts (1943):
  - Binary artificial neuron
  - Two inputs, one output
- Can be combined to make complicated expressions
- Below: output is active when at least two inputs are active



$C = A \wedge B$



$C = A \vee B$

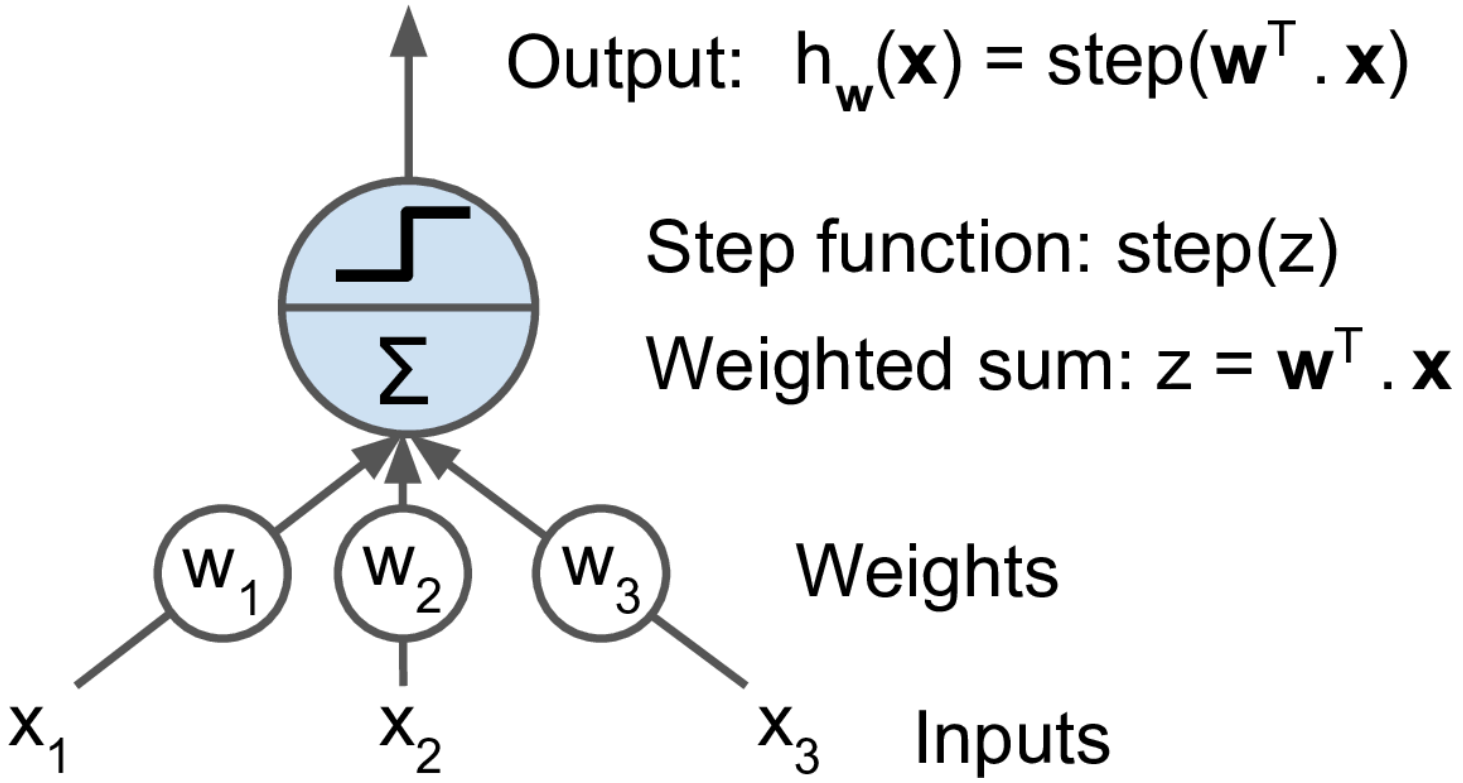


$C = B \wedge \sim A$

Perceptron

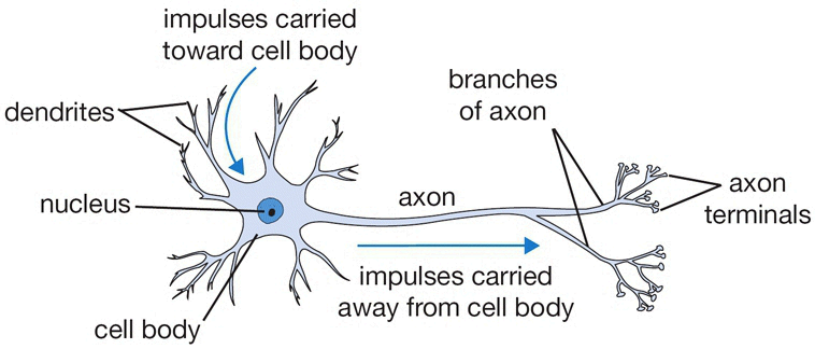
- Rosenblatt (1957): Now inputs can be floats
- Each input has an associated weight
- Weighted sum is put through a step (Heaviside) function
- $\text{heaviside(z)} = \begin{cases} 0 \; if \; z < 0 \\ 1 \; if \; z \geq 0 \end{cases}$
- $z = w_1 x_1 + w_2 x_2 + \; ... + w_n x_n$
- $h_w(z) = heaviside(z)$

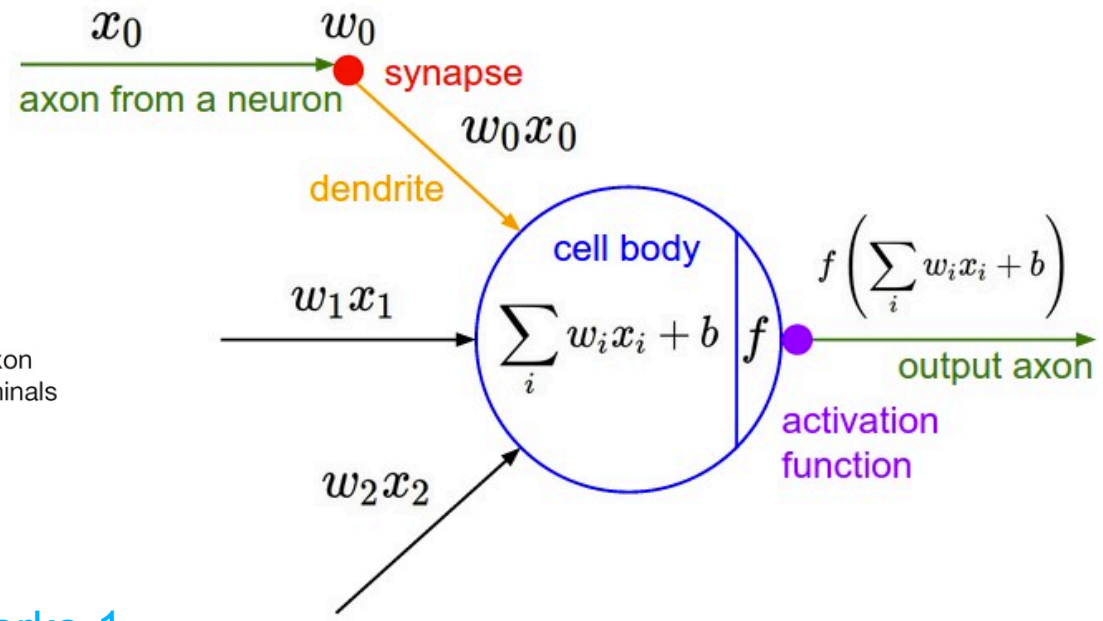
Perceptron


Output: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(\mathbf{w}^T . \mathbf{x})$
Step function: step(z)
Weighted sum: $z = \mathbf{w}^T . \mathbf{x}$
w_1
w_2
w_3
Weights
x_1
x_2
x_3
Inputs

Neurons

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





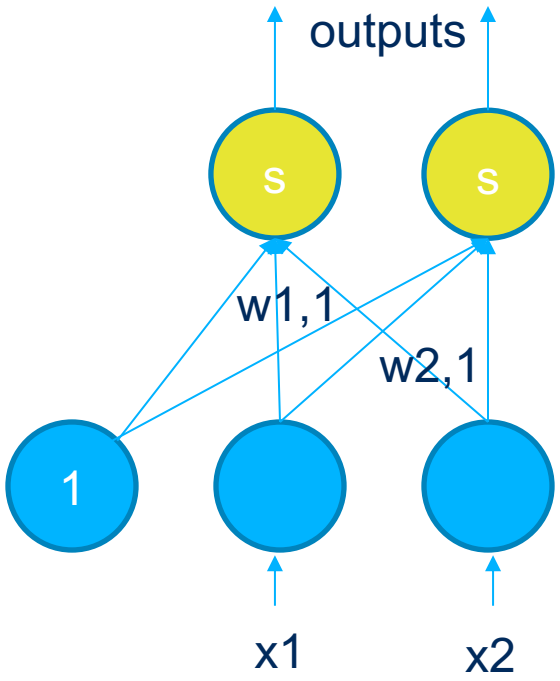
cs231n.github.io/neural-networks-1

Perceptron: A Linear Binary Classifier

- 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

Perceptron

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



Module 9 - Section 2

Training ANNs

Training a Perceptron

Adjust weights based on prediction error

$$w_{i,j} = w_{i,j} + n\left(\hat{y}_j - y_j\right)x_i \text{ where}$$

- $w_{i,j}$ connection weight between ith input and jth output
- x_i ith input
- $\hat{y}_j$ output of the of the jth neuron
- y_j target value

Training a Perceptron in Python

```
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]])
```

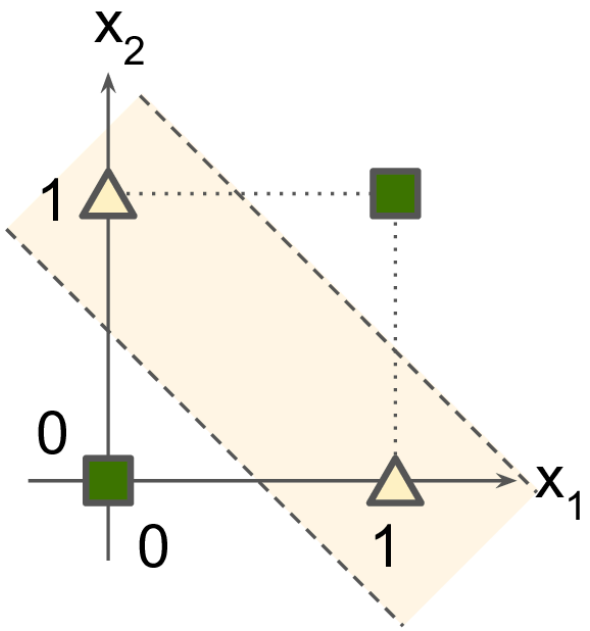
Fine Tuning Hyperparameters

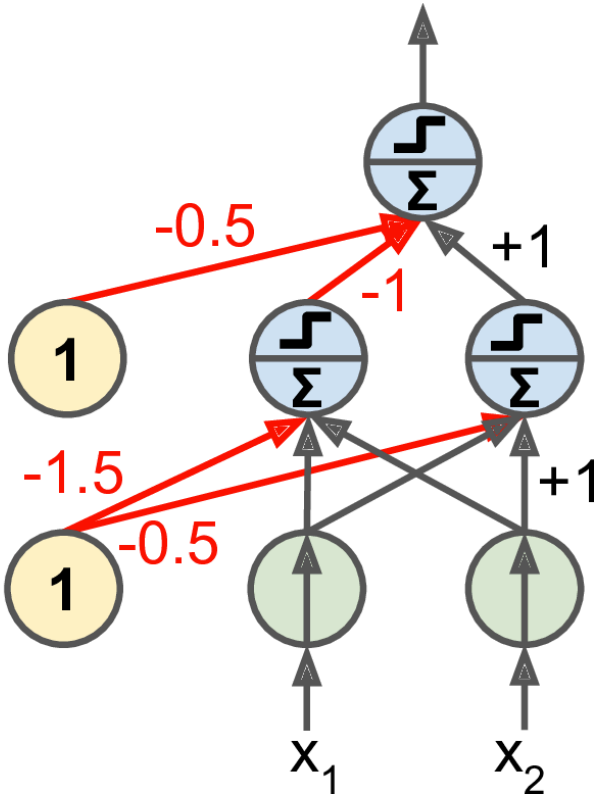
- In simple perceptron:
  - Number of layers
  - Number of neurons per layer
  - Type of activation function
  - Weight initialization

Module 9 - Section 3

Deep Neural Networks (DNNs)

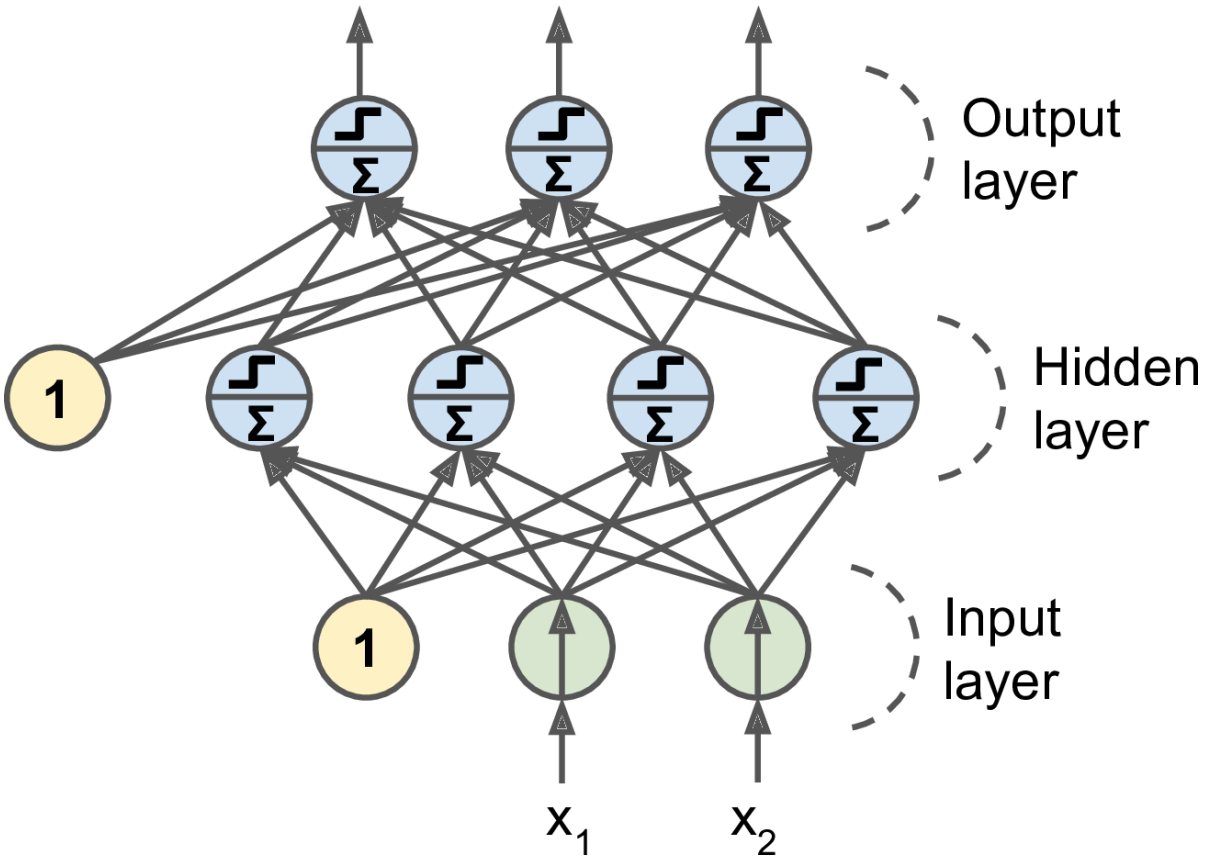
Multiple Layer Perceptron





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

Multiple Layer Perceptron



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 (e.g.,: face)

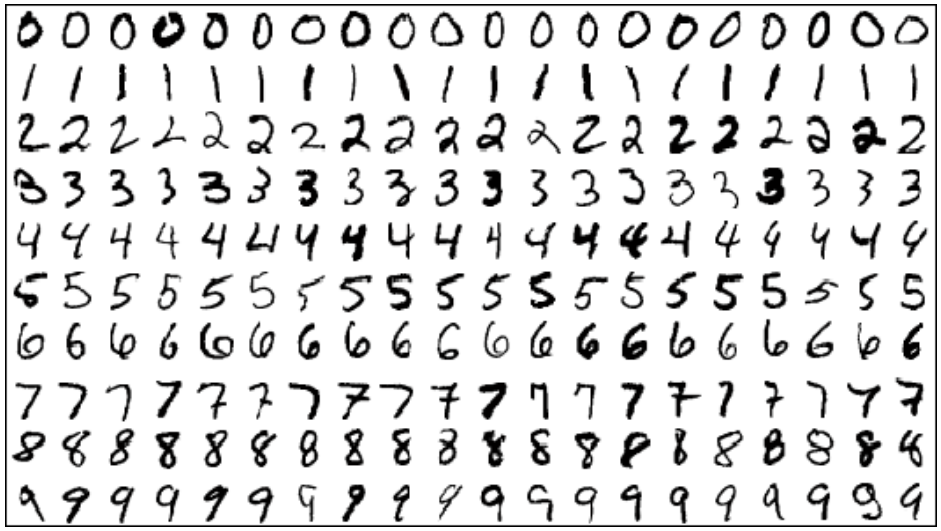
Number of Layers

- 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

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

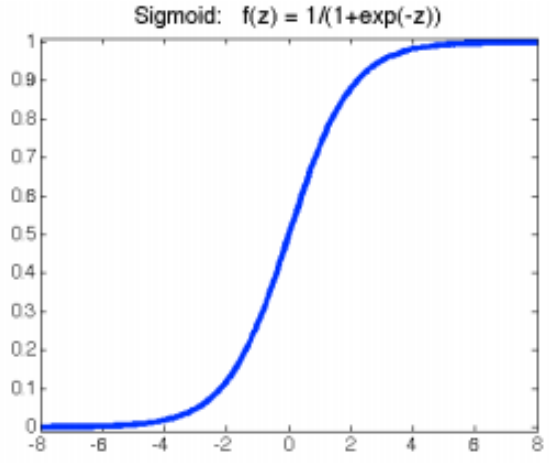
Activation Functions

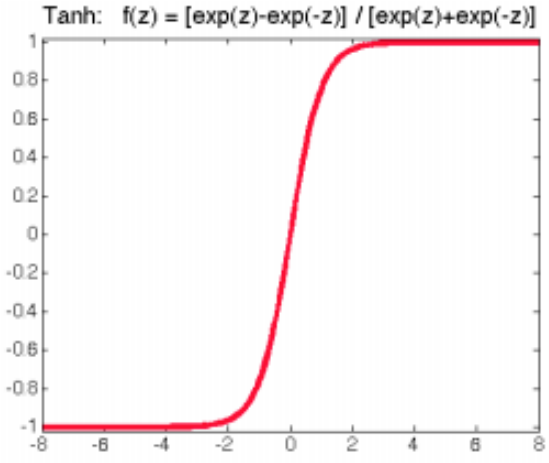
- Most commonly used activation functions:
  - Sigmoid
  - Tanh
  - ReLU (Rectifier Linear Unit)

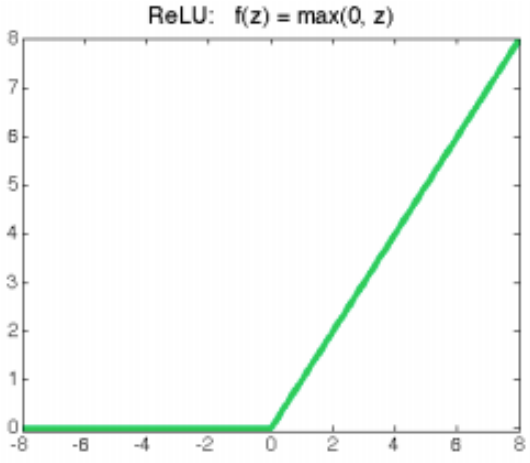
$$\sigma(z) = \frac{1}{1+\exp(-z)}$$

$$\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$$

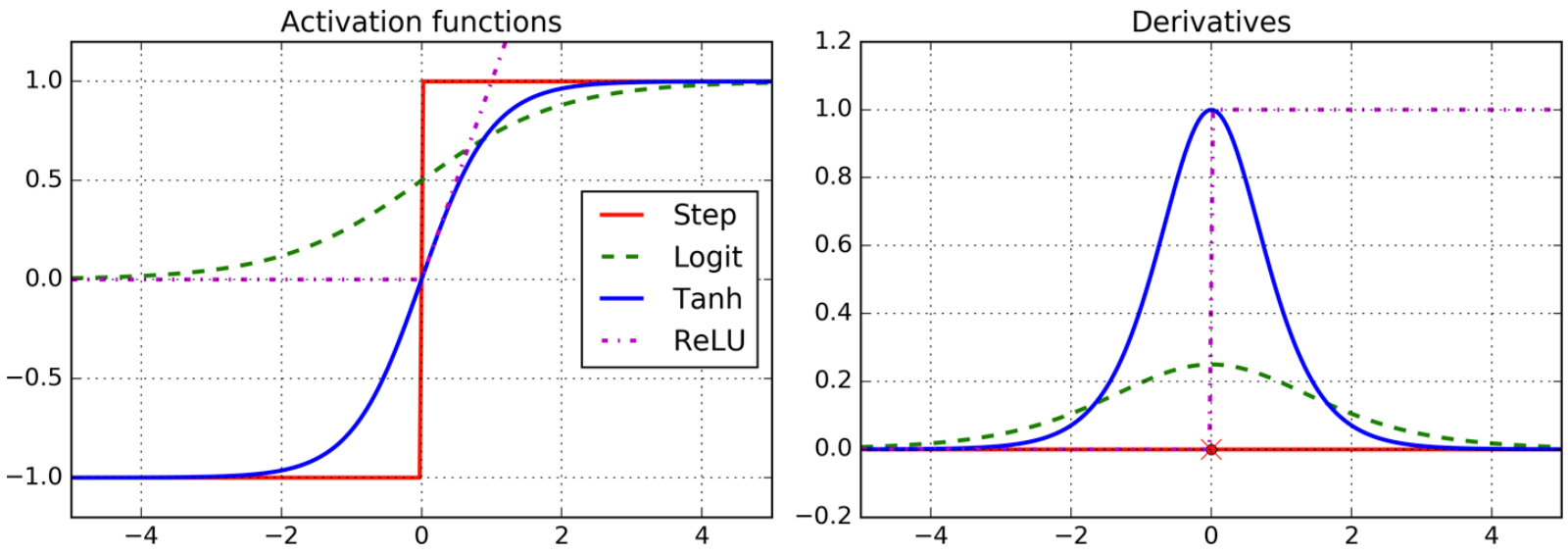
$$\mathrm{ReLU}(z) = \max(0, z)$$





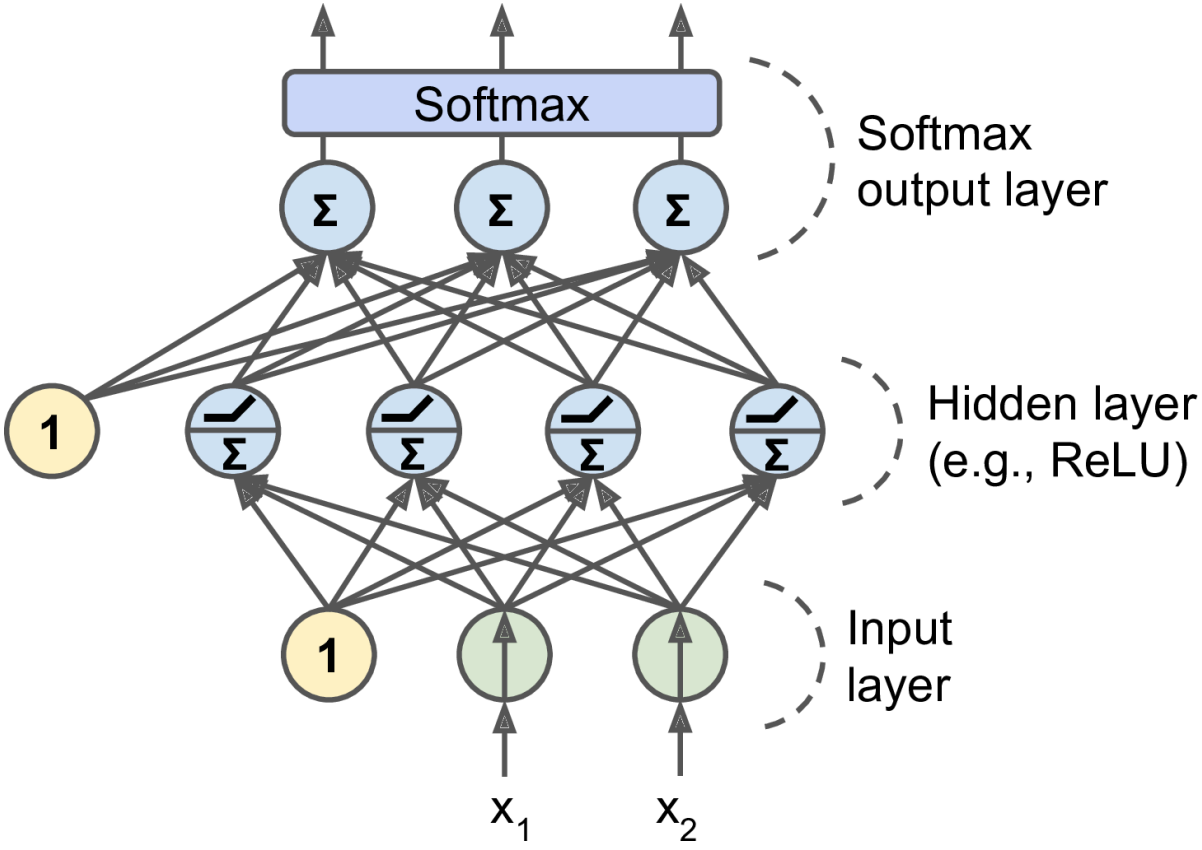


Activation Functions



Activation functions and their derivatives

MLP for Classification



Backpropagation

- For each training instance the backpropagation algorithm:
  - first makes a prediction (forward pass),
  - measures the error using a loss function,
  - 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 9 - Section 4

Training DNNs

Install TensorFlow 2.0

- Install the latest *TensorFlow* using *pip* on the command line
- Remember to activate your virtualenv if needed

- Run python and verify it was installed correctly

```
$ python3 -m pip install --upgrade tensorflow
```

```
>>> import tensorflow as tf
>>> from tensorflow import keras
>>> tf.__version__
'2.0.0'
>>> keras.__version__
'2.2.4-tf'
```

MLP with Sequential Dense Keras

- The simplest way to train an MLP with TensorFlow is to use the high-level Keras API
- The Sequential class makes it easy to train a deep neural network with any number of hidden layers, and a SoftMax output layer to estimate class probabilities
- Dense is a fully connected network architecture

```
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="relu"))
model.add(keras.layers.Dense(100, activation="relu"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

This would produce 98.2% accuracy on the test set!

MLP with Sequential Dense Keras

- Instead of adding layers one by one you can pass a list to the Sequential object directly

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

- It is helpful to store the hyperparameters that define the architecture in variables upfront

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

Keras Model Summary

- Use the summary method to review the model architecture is correct

```
>>> model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 300)               235500
_________________________________________________________________
dense_1 (Dense)              (None, 100)               30100
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
```

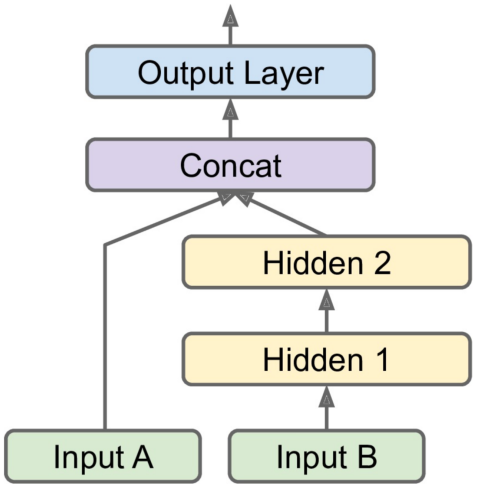
Keras Model Object

- All layers, names, weights and biases are stored in the model
- This allows easy access to help open the black box

```
>>> model.layers
[<tensorflow.python.keras.layers.core.Flatten at 0x132414e48>,
 <tensorflow.python.keras.layers.core.Dense at 0x1324149b0>,
 <tensorflow.python.keras.layers.core.Dense at 0x1356ba8d0>,
 <tensorflow.python.keras.layers.core.Dense at 0x13240d240>]
>>> hidden1 = model.layers[1]
>>> hidden1.name
'dense'
>>> model.get_layer('dense') is hidden1
True
>>> weights, biases = hidden1.get_weights()
>>> weights
array([[ 0.02448617, -0.00877795, -0.02189048, ..., -0.02766046,
         0.03859074, -0.06889391],
       ...,
       [-0.06022581,  0.01577859, -0.02585464, ..., -0.00527829,
         0.00272203, -0.06793761]], dtype=float32)
>>> weights.shape

(784, 300)
>>> biases
array([0., 0., 0., 0., 0., 0., 0., 0., 0., ...,  0., 0., 0.], dtype=float32)
>>> biases.shape
(300,)
```

Keras Functional API

- The functional API allows you to make non-sequential complex models
- This example allows the data to bypass the hidden layers so complex and simple patterns can be learned




```
input_A = keras.layers.Input(shape=[5], name="wide_input")
input_B = keras.layers.Input(shape=[6], name="deep_input")
hidden1 = keras.layers.Dense(30, activation="relu")(input_B)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1, name="output")(concat)
model = keras.Model(inputs=[input_A, input_B], outputs=[output])
```

Keras Subclassing API

- The final way to create a model is by defining a custom class based off the Model class
- The call function is all that is needed to define the layers connections

```
class WideAndDeepModel(keras.Model):
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs) # handles standard args (e.g., name)
        self.hidden1 = keras.layers.Dense(units, activation=activation)
        self.hidden2 = keras.layers.Dense(units, activation=activation)
        self.main_output = keras.layers.Dense(1)
        self.aux_output = keras.layers.Dense(1)

    def call(self, inputs):
        input_A, input_B = inputs
        hidden1 = self.hidden1(input_B)
        hidden2 = self.hidden2(hidden1)
        concat = keras.layers.concatenate([input_A, hidden2])
        main_output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return main_output, aux_output

model = WideAndDeepModel()
```

Model Training: Keras – SkLearn Similarities

- One of the nicest features of Keras is the similarity to SKLearn
- First you must compile the model to define the loss function and optimizer plus any metrics
- Then

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

```
>>> history = model.fit(X_train, y_train, epochs=30,
...                     validation_data=(X_valid, y_valid))
...
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [======] - 3s 49us/sample - loss: 0.7218     - accuracy: 0.7660
                                    - val_loss: 0.4973 - val_accuracy: 0.8366
```

Model Evaluation

- Evaluate the model with a test set

```
>>> model.evaluate(X_test, y_test)
10000/10000 [==========] - 0s 29us/sample - loss: 0.3340 - accuracy: 0.8851
[0.3339798209667206, 0.8851]
```

- Make predictions just like SKLearn but output depends on the activation function, in this case probabilities

```
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.03, 0.  , 0.01, 0.  , 0.96],
       [0.  , 0.  , 0.98, 0.  , 0.02, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

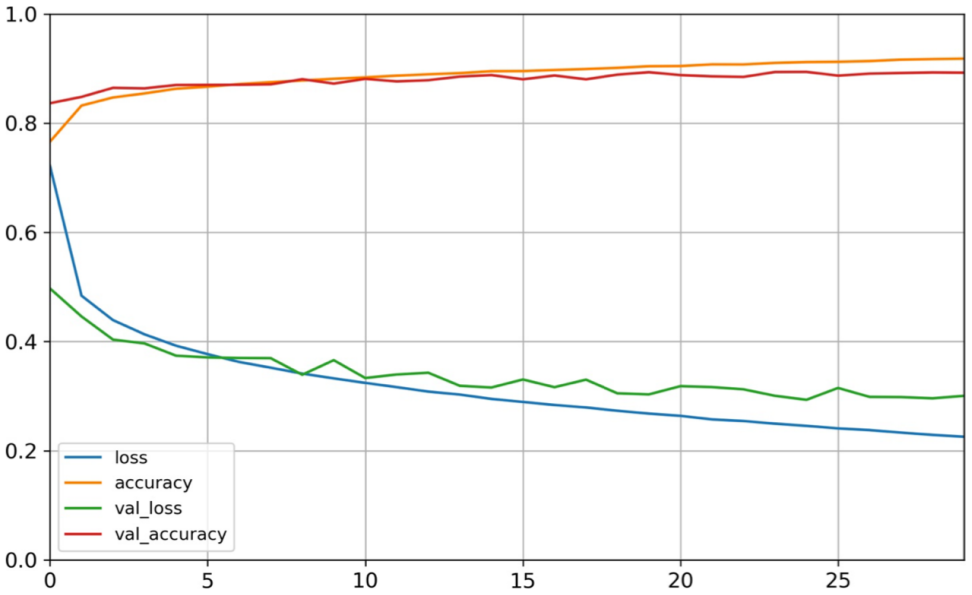
- Save and load models in HDF5 format

```
model.save("my_keras_model.h5")

model = keras.models.load_model("my_keras_model.h5")
```

Model Training

- The history can be used to plot the training curves directly since all epoch results are stored

```
import pandas as pd
import matplotlib.pyplot as plt

pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1) # set the vertical range
plt.show()
```

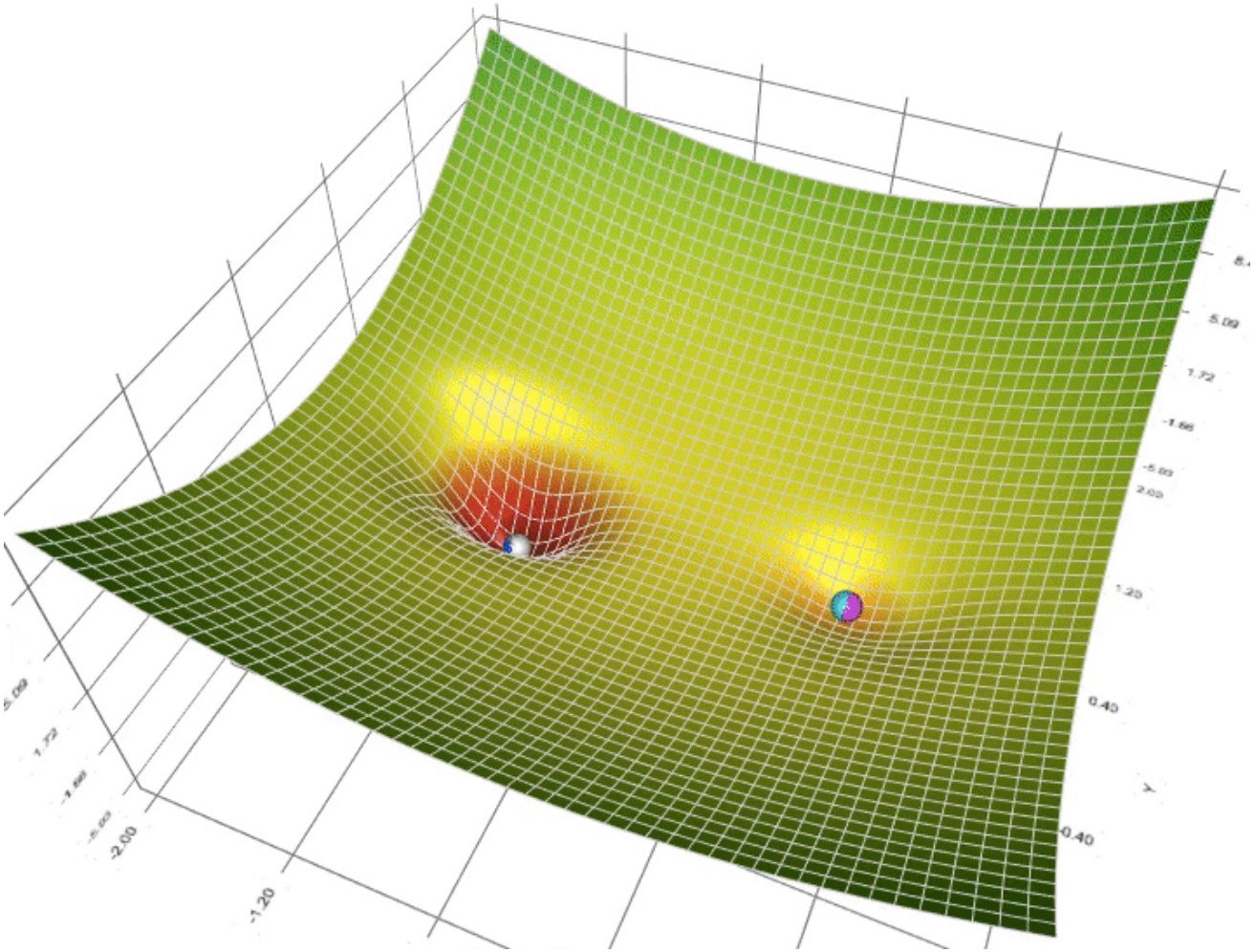


Module 9 - Section 5

Optimizers

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 optimizers:
  - Momentum
  - Nesterov Accelerated Gradient
  - AdaGrad
  - RMSProp
  - Adam



Animation of 5 gradient descent methods on a surface: gradient descent (cyan), momentum (magenta), AdaGrad (white), RMSProp (green), Adam (blue). Left well is the global minimum; right well is a local minimum. Full article here

Momentum Optimization

- Imagine a bowling ball rolling down a gentle slope on a smooth surface:
  - it will start out slowly, but it will quickly pickup momentum until it eventually reaches maximum possible 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 momentum is used as an acceleration, not as a speed.

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

$$2. \quad \theta \leftarrow \theta + \mathbf{m}$$

- Beta is momentum and m is momentum vector

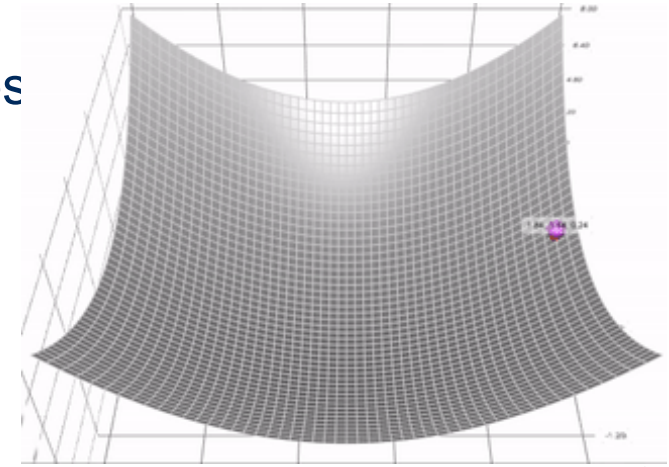
Momentum Optimization

- 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 = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

there is no friction; has only momentum.

- The bigger momentum, the faster it oscillates
- We use beta as decay-rate to slow down



Momentum Optimization

-The surface is our cost function

-Red area is global min

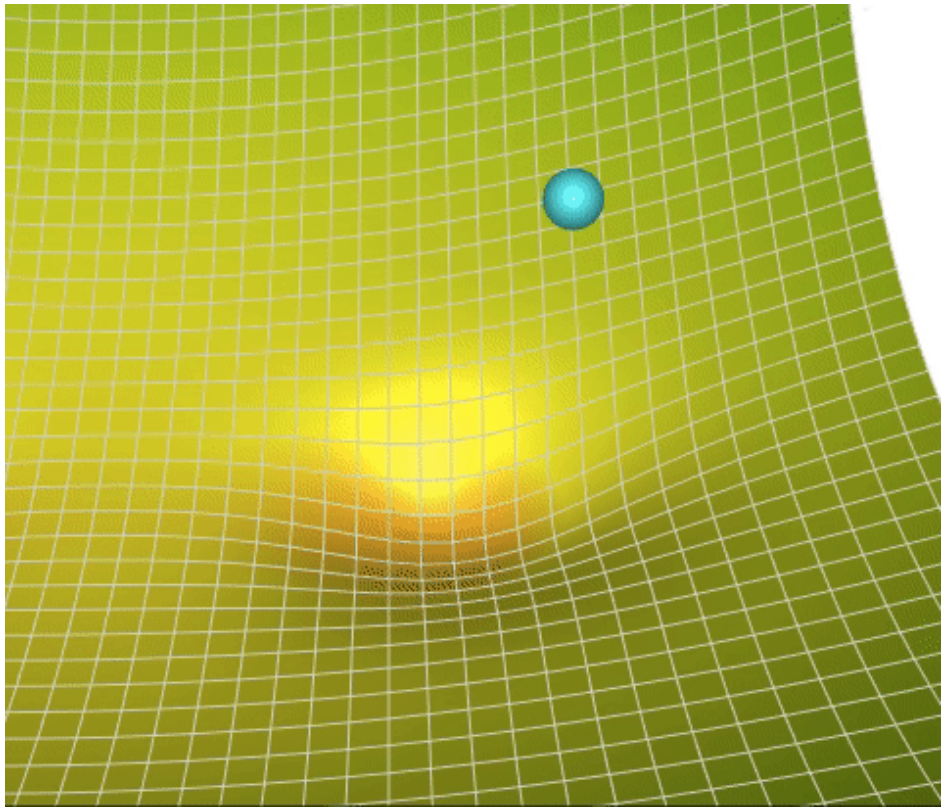
-Cyan color vectors are partial derivatives at every point (since there are 2 features)

-Black vector is the direction of next step

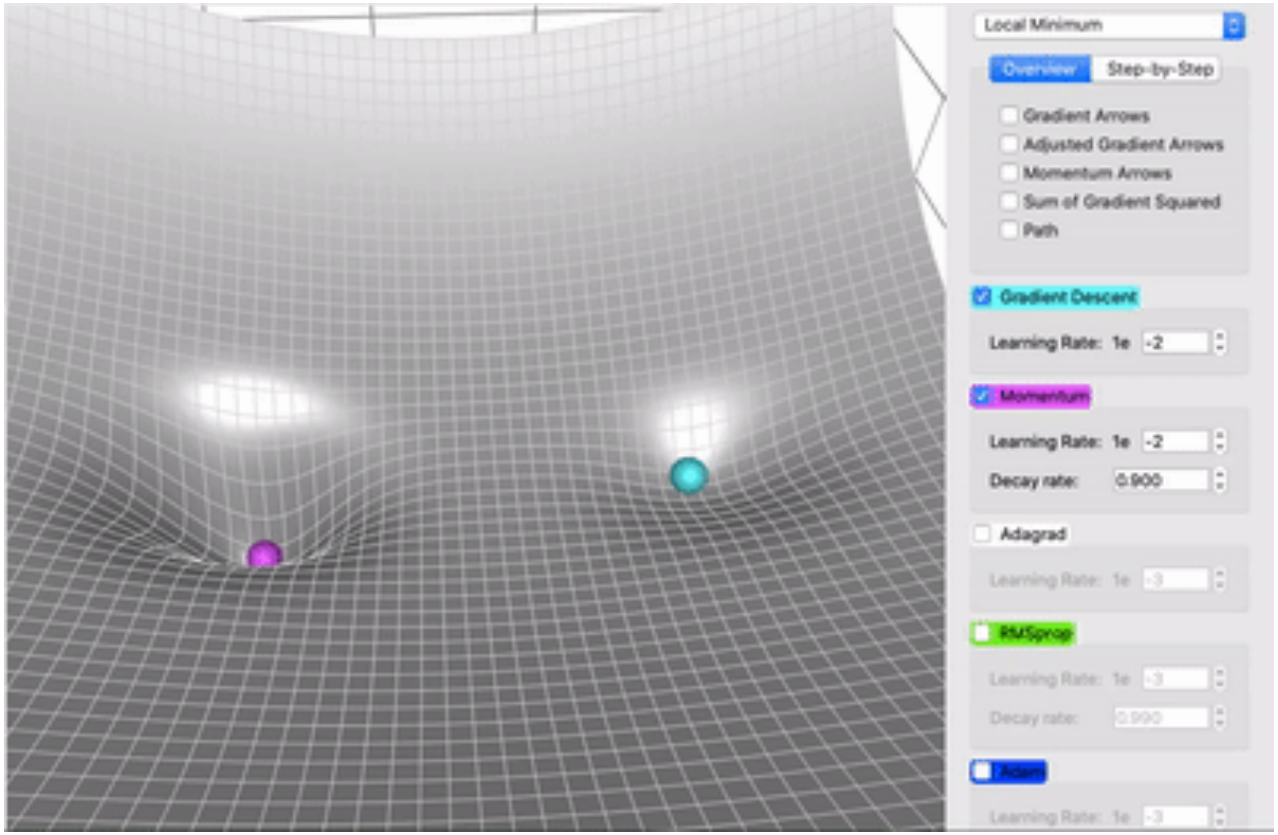
Benefits?

- Faster to converge

- Skips local minimum because of momentum



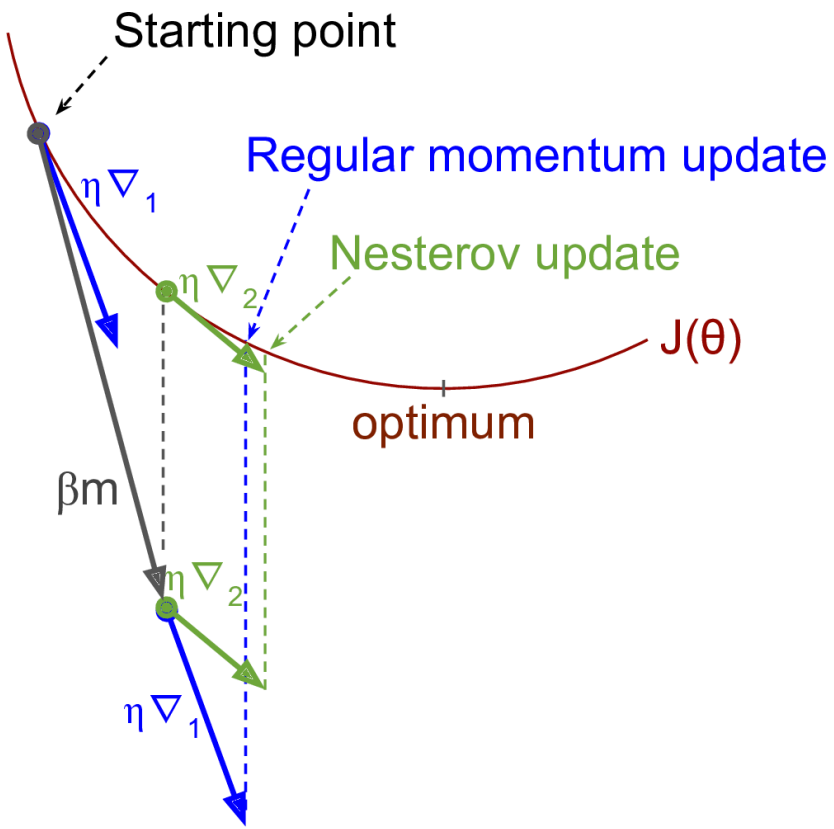
Vanilla GD vs Momentum GD



Momentum (magenta) vs. Gradient Descent (cyan) on a surface with a global minimum (the left well) and local minimum (the right well)

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 = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

AdaGrad (Adaptive Gradient)

- 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

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

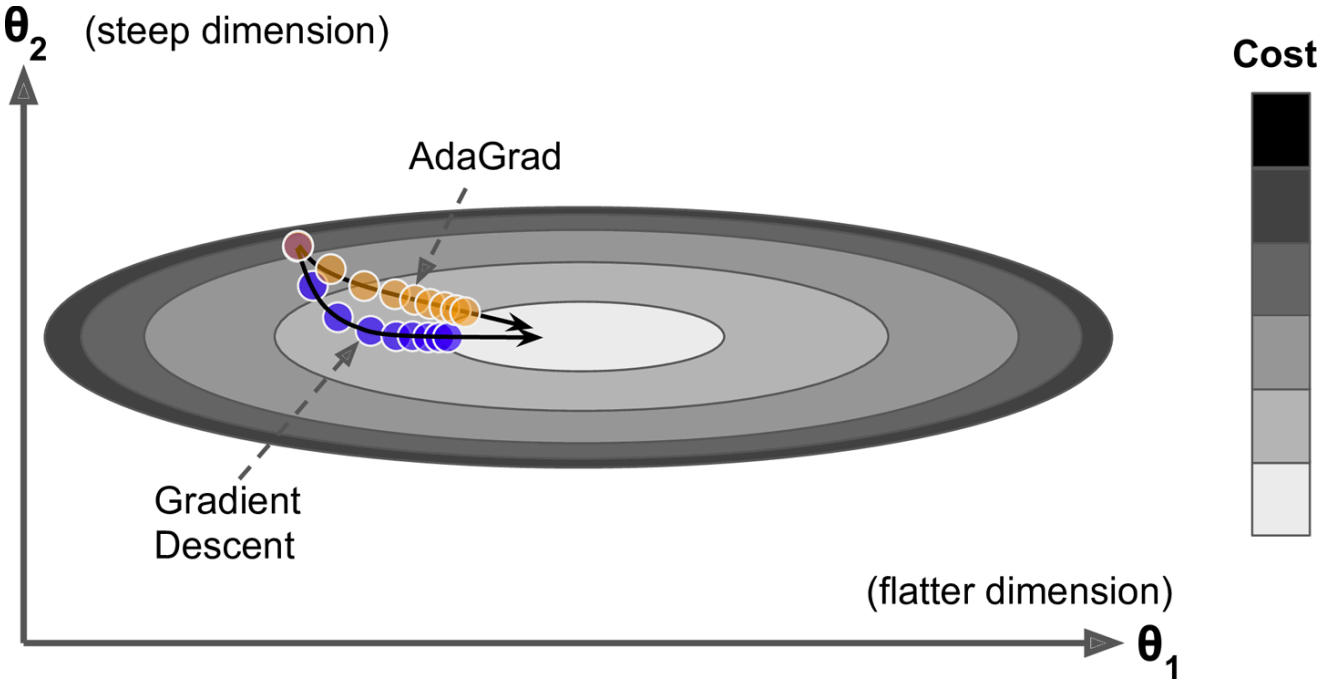
$$2. \quad \theta \leftarrow \theta - \eta \, \nabla_\theta J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}$$

- 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

$\otimes$ is Tensor product and $\oslash$ is element wise division

AdaGrad

- 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 = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

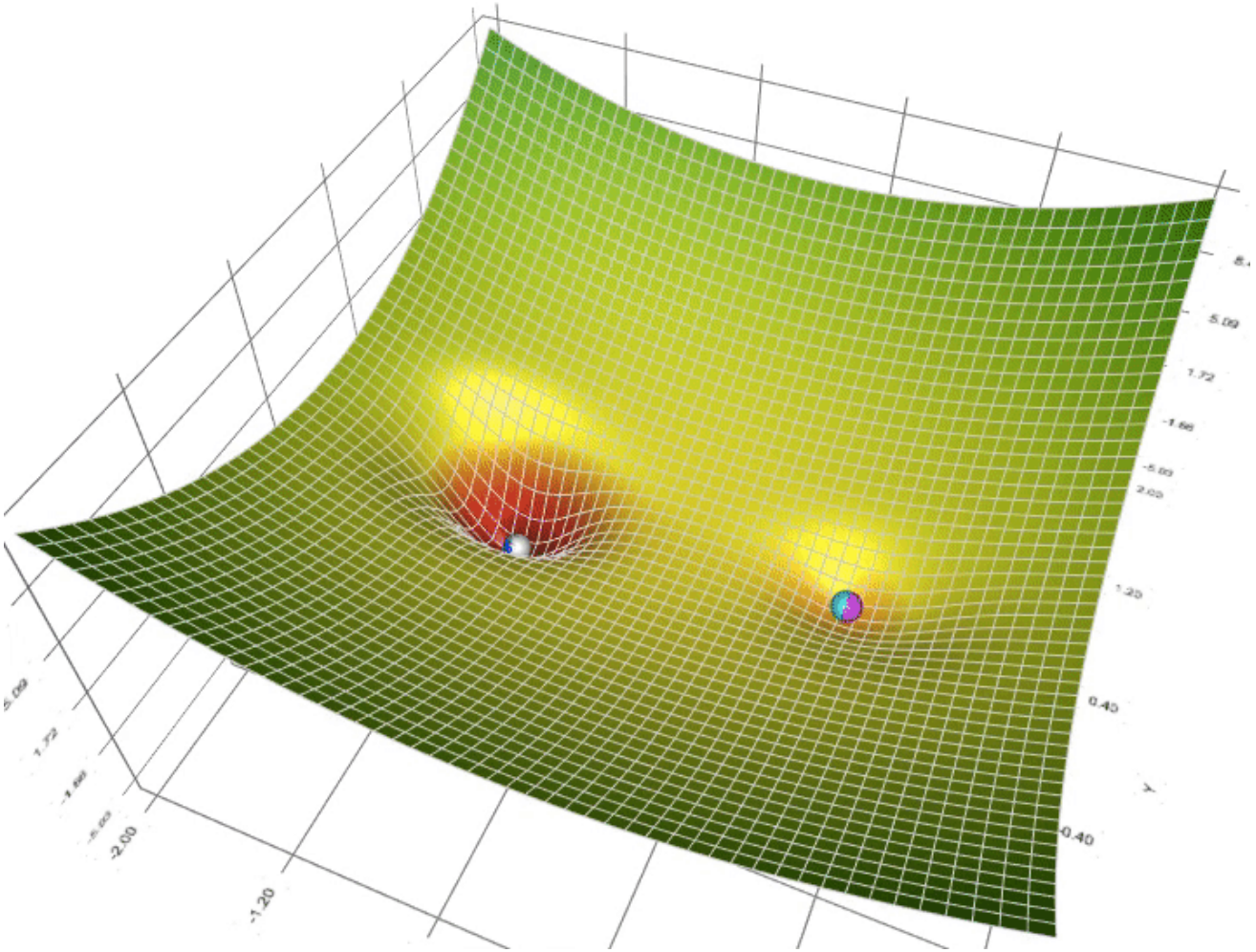
Adam Optimization

- *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 = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```



Animation of 5 gradient descent methods on a surface: gradient descent (cyan), momentum (magenta), AdaGrad (white), RMSProp (green), Adam (blue). Left well is the global minimum; right well is a local minimum. Full article here

How to use Optimizer?

- Define your network architecture

```
model = keras.models.Sequential([
        keras.layers.Flatten(input_shape=[28, 28]),
        keras.layers.Dense(300, activation="relu"),
        keras.layers.Dense(100, activation="relu"),
        keras.layers.Dense(10, activation="softmax")])
```

- Compile model with optimizer

```
model.compile(loss="sparse_categorical_crossentropy",
optimizer=keras.optimizers.SGD(lr=1e-3),
metrics=["accuracy"])
```

- Start fitting with X and Y

```
history = model.fit(X_train, y_train, epochs=10,
validation_data=(X_valid, y_valid))
```

Summary

Class	Convergence speed	Convergence quality
`SGD`	*	***
`SGD(momentum=...)`	**	***
`SGD(momentum=..., nesterov=True)`	**	***
`Adagrad`	***	* (stops too early)
`RMSprop`	***	** or ***
`Adam`	***	** or ***
`Nadam`	***	** or ***
`AdaMax`	***	** or ***

WARNING

Adaptive optimization methods (including RMSProp, Adam, and Nadam optimization) are often great, converging fast to a good solution. However, a 2017 paper[20] by Ashia C. Wilson et al. showed that they can lead to solutions that generalize poorly on some datasets. So when you are disappointed by your model's performance, try using plain Nesterov Accelerated Gradient instead: your dataset may just be allergic to adaptive gradients. Also check out the latest research, because it's moving fast.

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Module 9 - Section 6

Training Challenges

Deep Learning Challenges

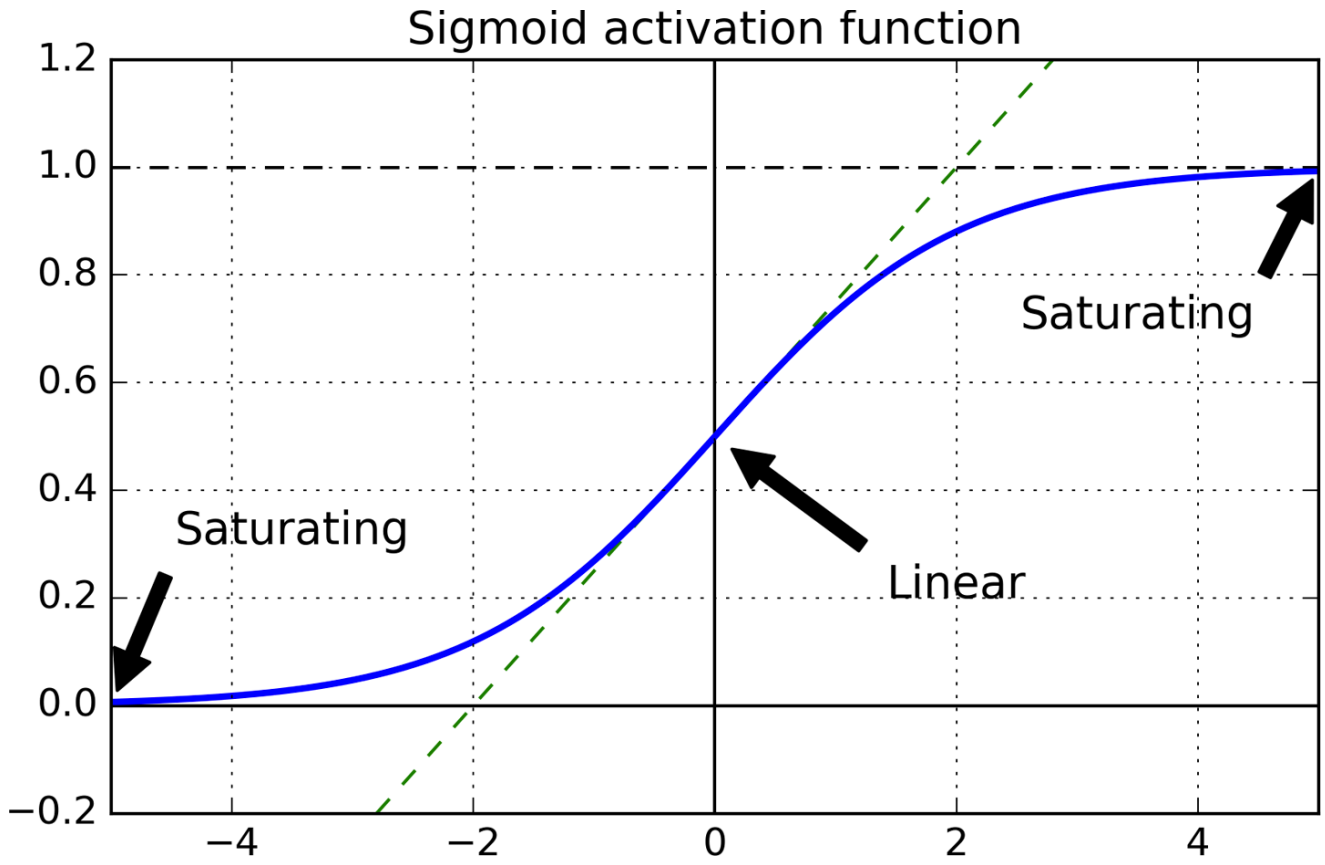
- V*anishing gradients* problem (or the related *exploding gradients* problem) that affects deep neural networks and makes lower layers very hard to train.
- For large networks, training can be extremely slow.
- A model with millions of parameters would severely risk overfitting the training set.

Output Layer Error	$E_o = (O - y) \cdot R'(Z_o)$
Hidden Layer Error	$E_h = E_o \cdot W_o \cdot R'(Z_h)$

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



Vanishing/Exploding Gradients

- 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

- By default Keras uses Xavier initialization but it can be changed to *he_uniform* or *he_normal* like so:

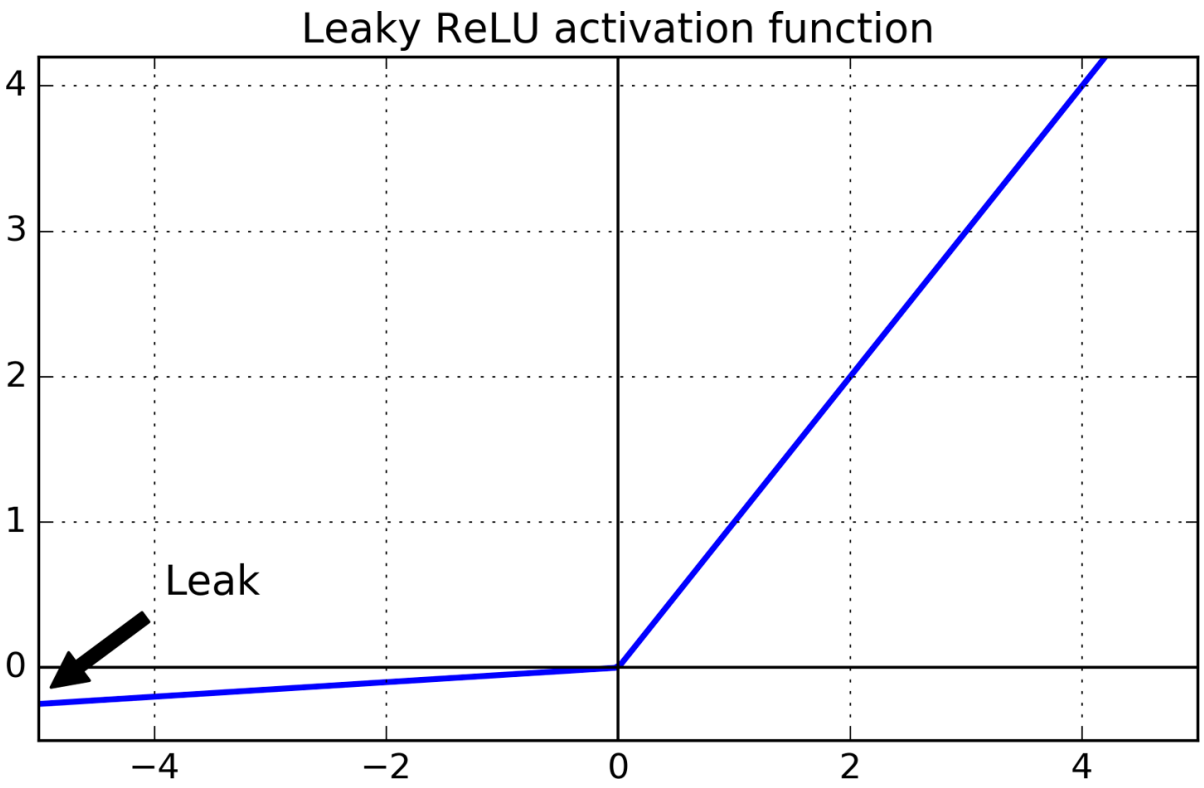
Activation function	Uniform distribution [–r, r]	Normal distribution
Logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$
Hyperbolic tangent	$r = 4\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{inputs}}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

```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

Non-Saturating Activation Functions

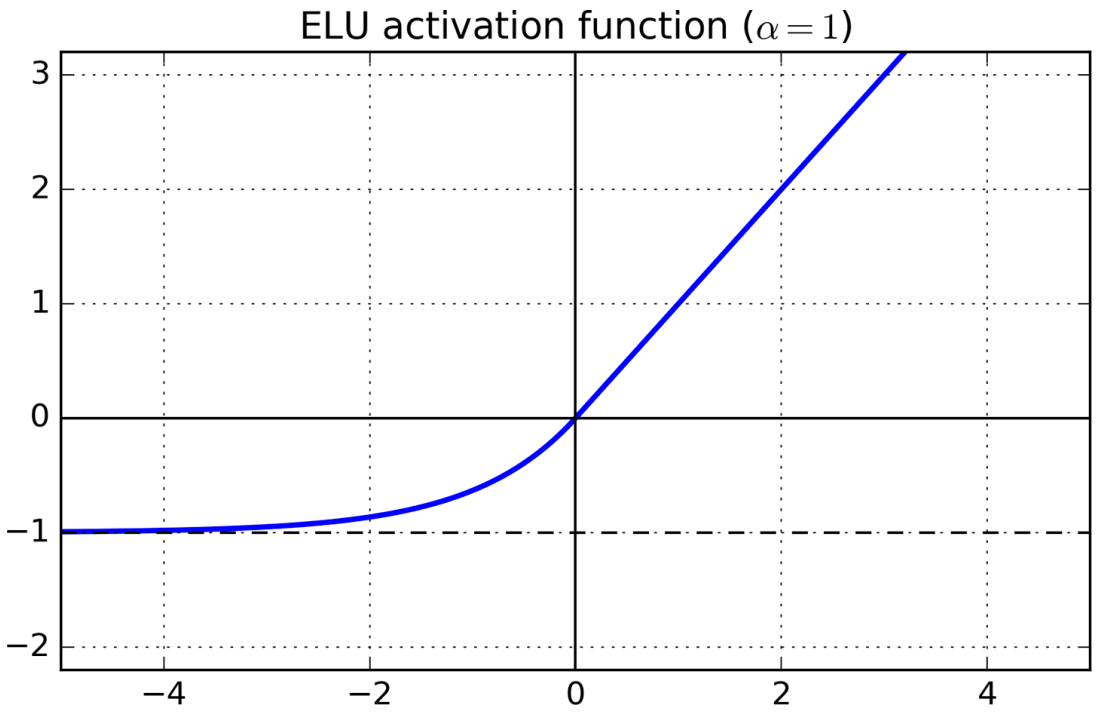
- ReLU does not saturate for positive values
  - $relu(z) = \max(0, z)$
- Has problem of *dying ReLUs*: during training, some neurons begin only outputting zero
  - If weighted sum of neuron's inputs is negative, this will happen
  - Gradient of ReLU function is zero when input is negative
- Solution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

Leaky ReLU




```
model = keras.models.Sequential([
    [...]
    keras.layers.Dense(10, kernel_initializer="he_normal"),
    keras.layers.LeakyReLU(alpha=0.2),
    [...]
])
```

Exponential Linear Unit



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

```
layer = keras.layers.Dense(10, activation="selu",
                           kernel_initializer="lecun_normal")
```

- Outperforms all ReLU variants:
  - training time reduced
  - better accuracy on test set
- Main drawback is that it is slower to compute during inference (prediction)

- A newer variant SELU is a scaled ELU that tends to self normalize, solving the vanishing/exploding gradient problem
- Requires LeCun initialization and sequential architecture to work

Which Activation Function to Choose

- In general
  - SELU > ELU > leaky ReLU (and its variants) > ReLU > tanh > logistic
  - If you care a lot about runtime performance, then you may prefer leaky ReLUs over SELU/ELU
  - 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 SELU/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 9 - Section 7

Training Strategies

General Strategy

- Build a neural network that can overfit the data
- Reduce the overfitting to acceptable levels
- Steps to reducing overfitting in order of importance:
  1. Get more data to provide more training examples
  2. Do data augmentation on the data you have
  3. Change model to a generalizable one (ensemble approaches)
  4. Add regularization to the model
  5. Reduce the model complexity to simplify

Batch Normalization

- 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
- $\mu_B = \frac{1}{m}\sum_{i=1}^{m_B} x^{(i)}$ $\quad\quad$ $\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m_B}\left(x^{(i)} - \mu_B\right)^2$
- $\hat{x}^{(i)} = \frac{x^{(i)} - \mu_B}{\sqrt{{\sigma_B}^2 + \epsilon}}$ $\quad\quad$ $z^{(i)} = \gamma\ \hat{x}^{(i)} + \beta$
- 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

Batch Normalization

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, kernel_initializer="he_normal", use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
    keras.layers.Dense(100, kernel_initializer="he_normal", use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
    keras.layers.Dense(10, activation="softmax")
])
```

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

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

- In general, batch normalization is preferred and is almost always used nowadays

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

- Simply add a *kernel_regularizer* hyperparameter to any layer definition

```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))
```

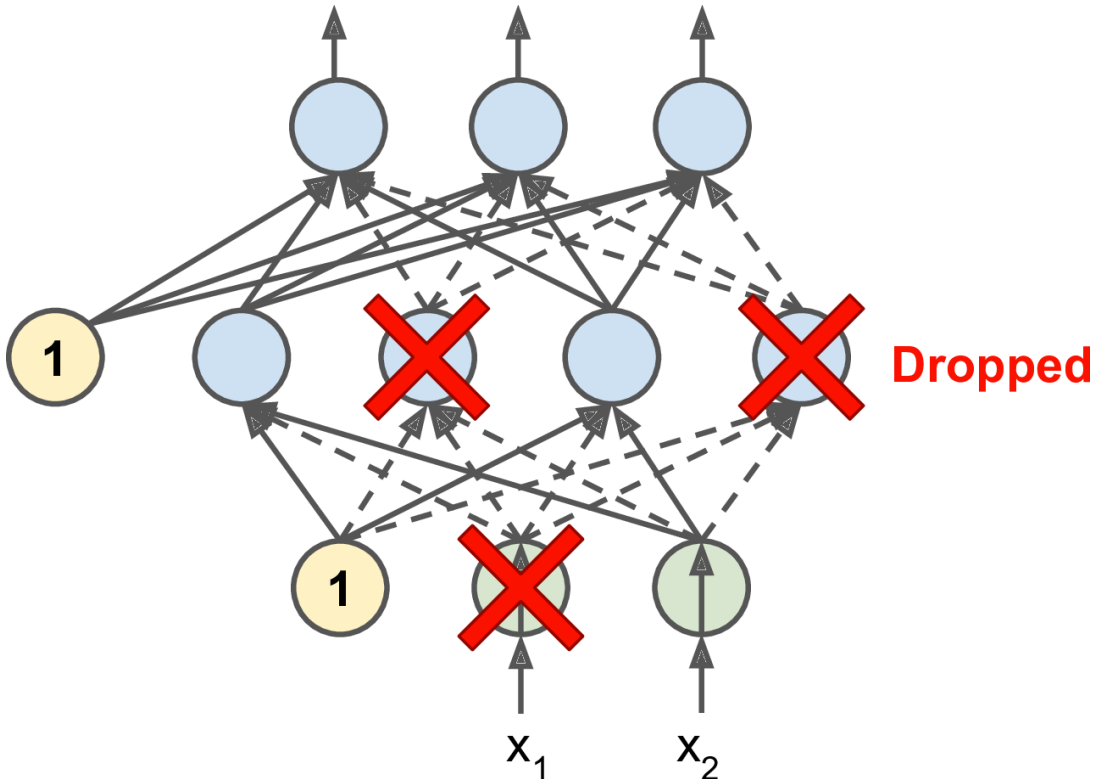
- Valid options include:
  - `l1()`
  - `l2()`
  - `l1_l2()`

Dropout

- 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.

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```

Dropout



WARNING

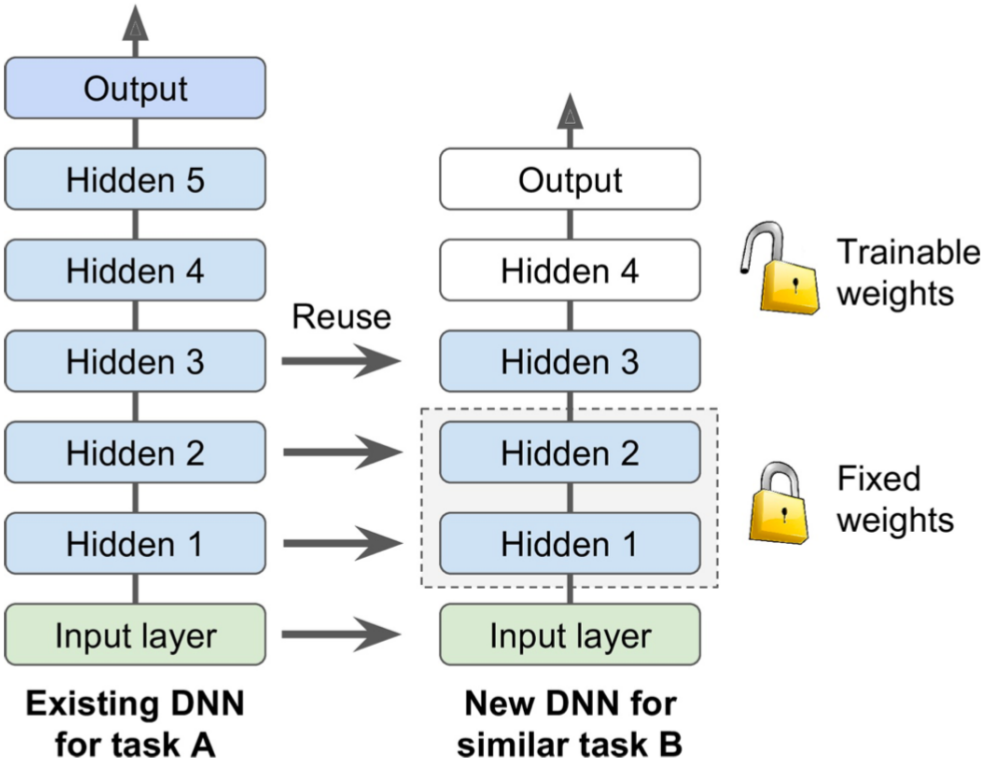
Since dropout is only active during training, comparing the training loss and the validation loss can be misleading. In particular, a model may be overfitting the training set and yet have similar training and validation losses. So make sure to evaluate the training loss without dropout (e.g., after training).

Pretrained Layers

- 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

Pretrained Layers

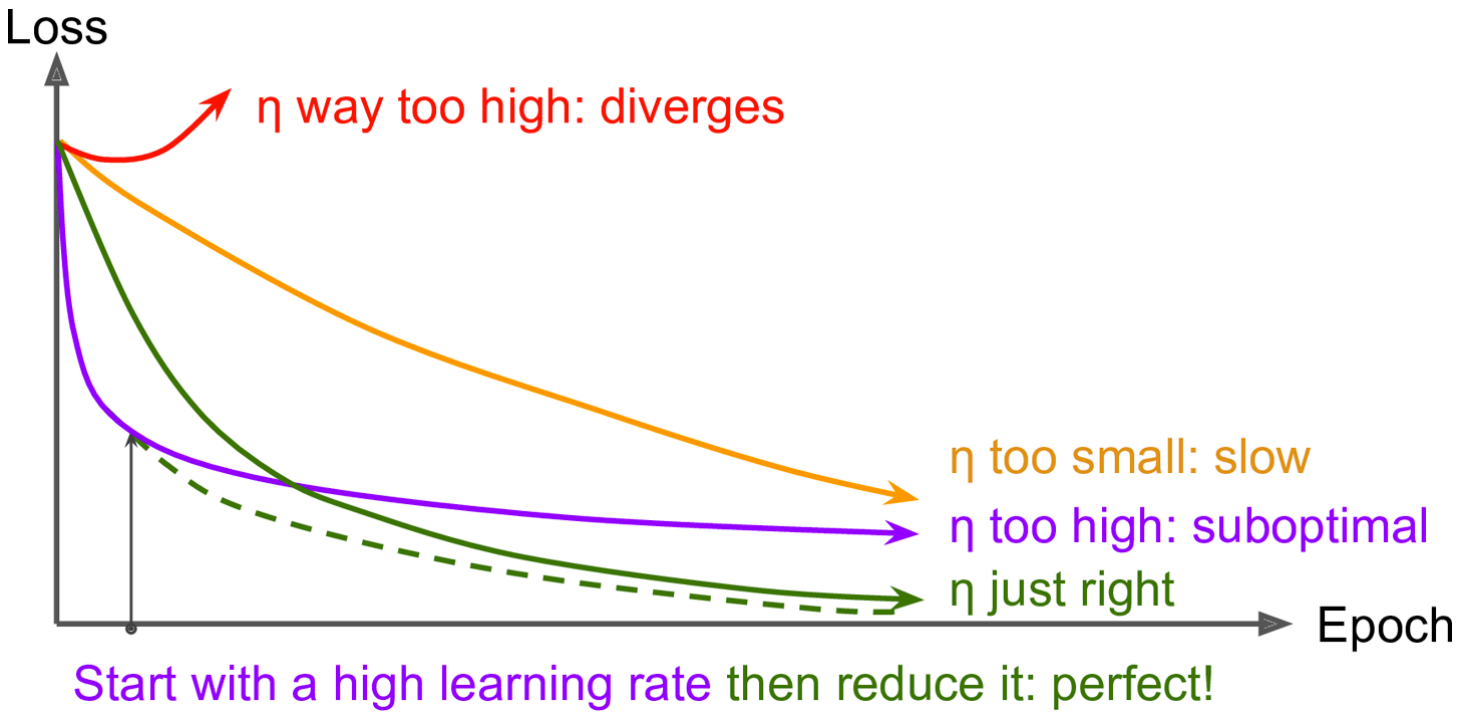
```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))

for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False
```



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




```
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train_scaled, y_train, [...], callbacks=[lr_scheduler])
```

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

Suggested Configuration

Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ELU
Normalization	None if shallow; Batch Norm if deep
Regularization	Early stopping (+ℓ_2 reg. if needed)
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

Suggested Configuration (If Sequential Only)

Hyperparameter	Default value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle

Module 9 - Section 8

Wrap-up

Homework

- Keep working on your Term Project!

Next Class

- Introduction to TensorFlow
  - Created by Google: the most popular deep learning framework in use today
- Read textbook chapters 9 & 10 in preparation

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Thank You

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

Join the conversation with us online:

facebook.com/uoftscs

@uoftscs

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

@uoftscs

UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

Thank You

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