AI: TensorFlow Neural Network

CPSC 501: Advanced Programming Techniques
Fall 2020

Jonathan Hudson, Ph.D
Instructor
Department of Computer Science
University of Calgary

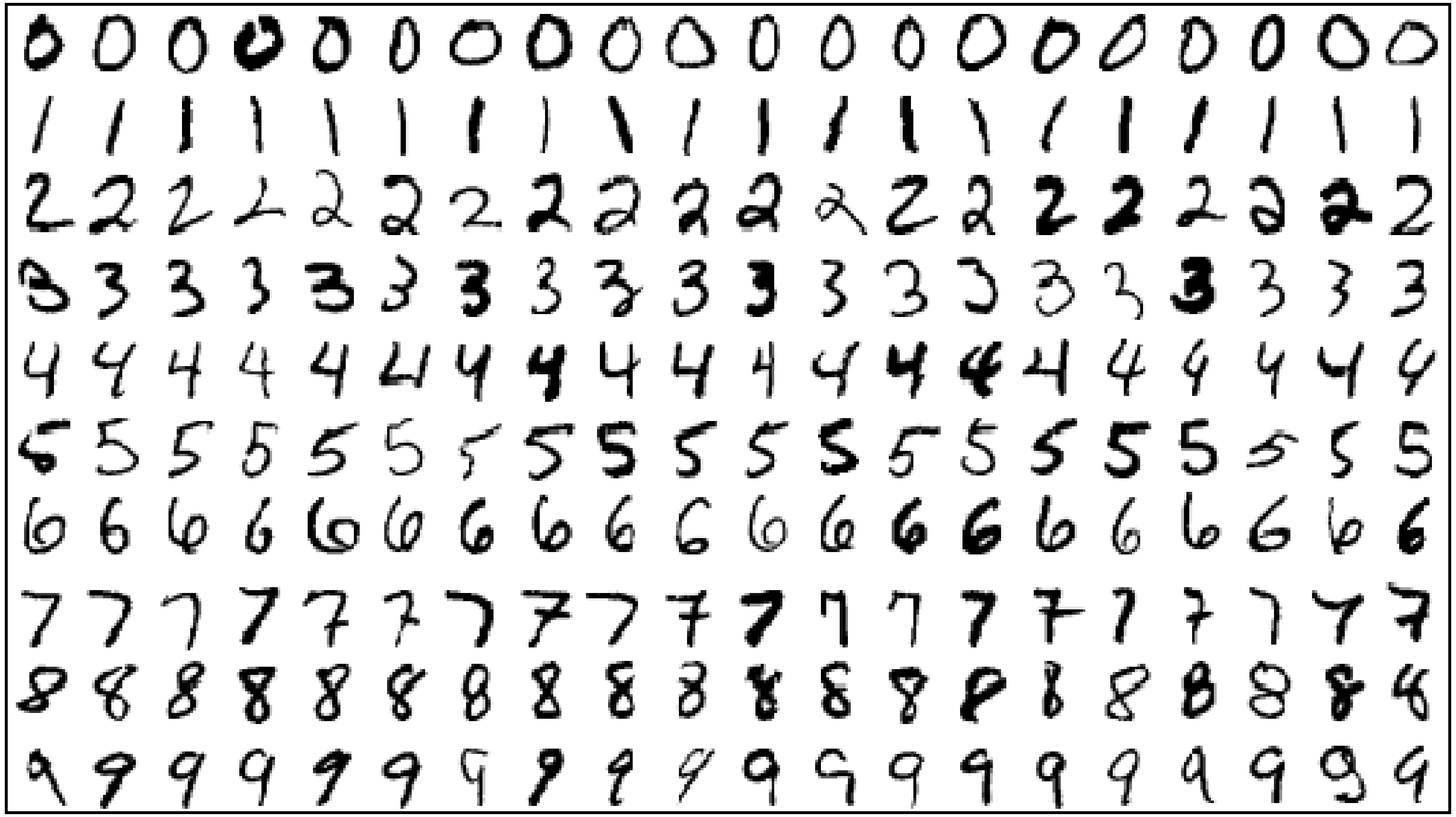
Wednesday, August 12, 2020

UNIVERSITY OF CALGARY

MNIST

One MNIST Database

- Each image is a 28x28 array, flattened out to be a 1-d tensor of size 784



Model

- Input to model
- X: image of a handwritten digit
- Y: the digit value

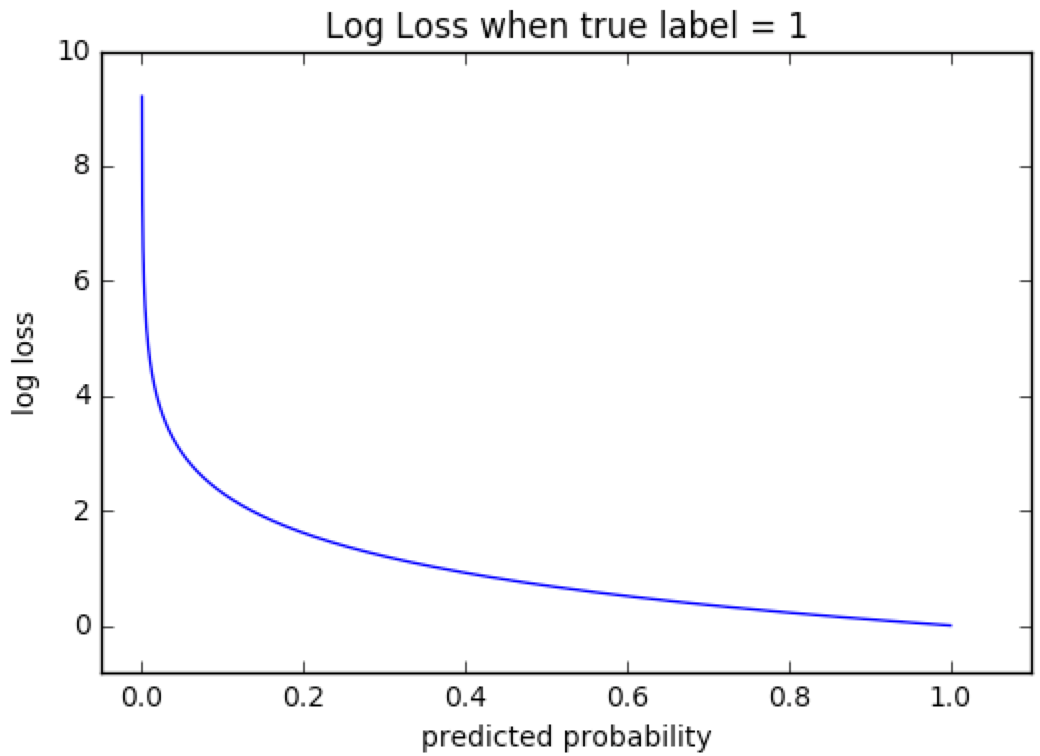
- Goal: trained model that recognizes the digit in the image

Model

- **Inference**: Y_predicted = softmax(X * w + b)
  - We want network that predicts 10 digits
  - We also want the sum of our probabilities across output layer to be 1
  - Sigmoid activation would give use between 0 and 1
  - Softmax goes step further and makes sure sum of the 10 probabilities are 1 in total

Model

- **Cross entropy loss:** -log(Y_predicted)
  - Made for measuring performance of models where output is 0 to 1



Pre-process

Process data

#TF2 Includes MNIST data already (mostly for learning purposes)

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()

#We need to level color data to 0 to 1 range

x_train, x_test = x_train / 255.0, x_test / 255.0

#We are classifying digits 0 to 9
class_names = list(range(10))

Graph (Neural Network)

Phase 1: Assemble our graph

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

Two layers

1. First we flatten image 2d array to a 1d tensor input
2. Then we make a connection from every image spot to every 0-9 integer output spot

Optimizer

Specify loss function

```
model.compile(
        optimizer='adam’,
        loss='sparse_categorical_crossentropy’,
        metrics=['accuracy’])
```

Use ‘adam’ optimizer

We’ll discuss the loss function later in slides

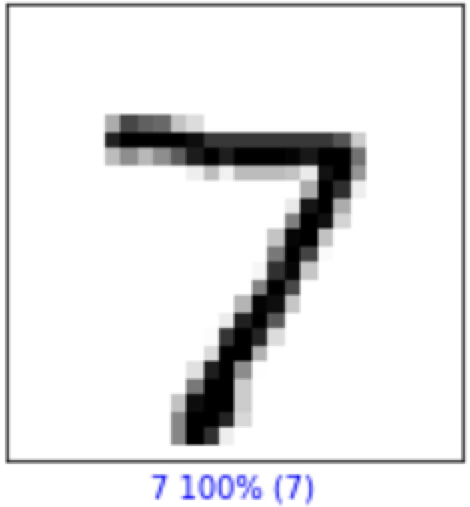
Train

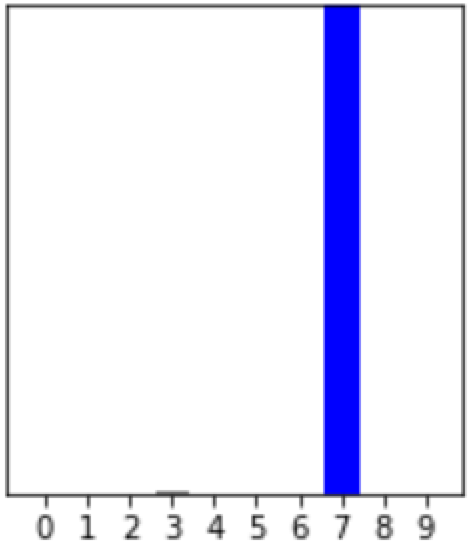
Train our model and evaluate it’s quality

```
model.fit(x_train, y_train, epochs=5)

model_loss, model_acc = model.evaluate(x_test,  y_test, verbose=2)

print(f"Model Loss: {model_loss*100:.1f}%")
print(f"Model Accuray:{model_acc*100:.1f}%")
```

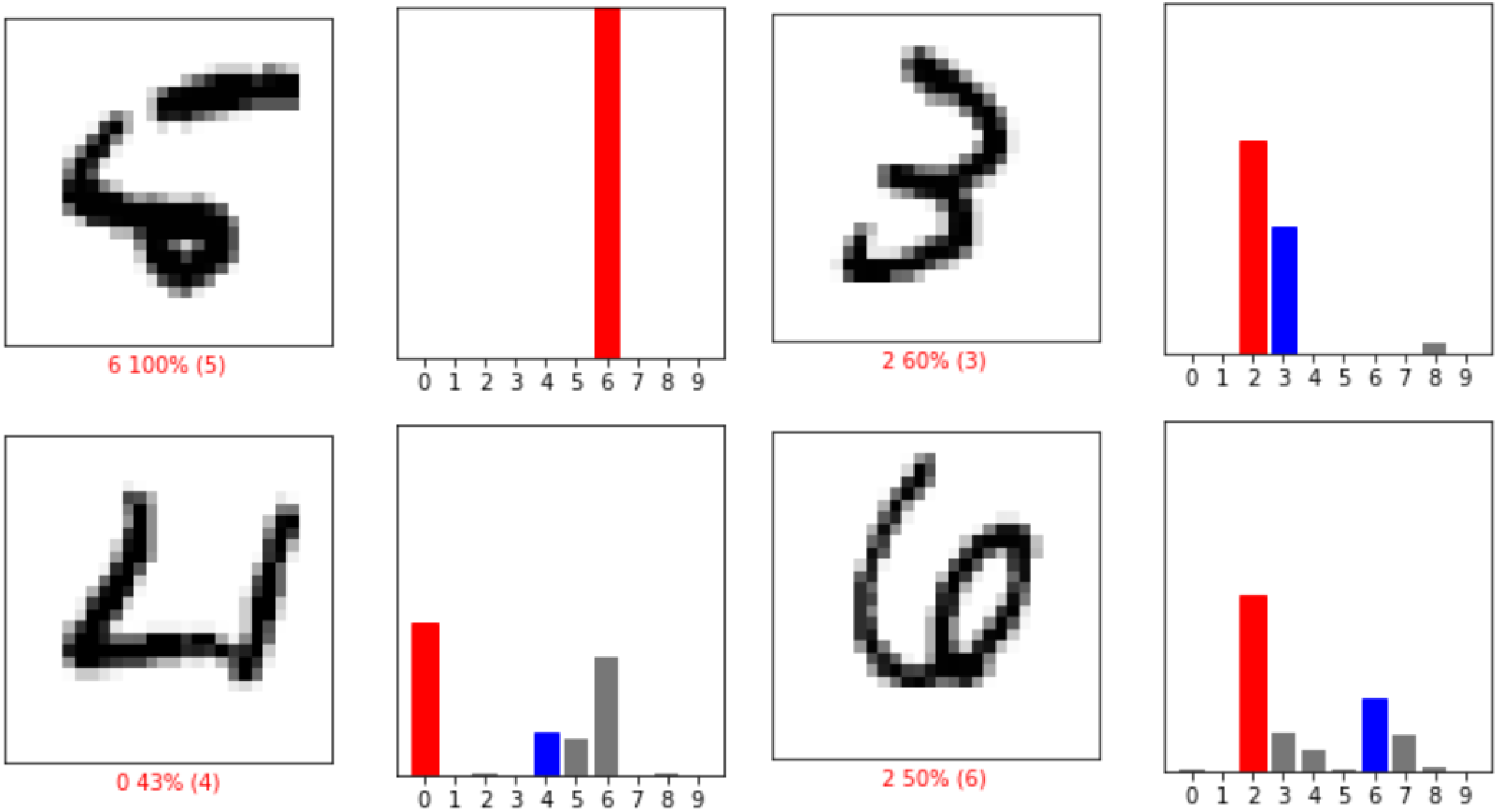
Output

Train our model and evaluate it's quality





Train our model and evaluate it’s quality



Saving and Loading

Save/Load our model

model.save('MINST.h5')

new_model = tf.keras.models.load_model('MINST.h5')

Can use this model exactly the same way we were the one we made and trained

How most apps works. Make model on the development end, spend a bunch of time testing it in dev, once the accuracy is good see if size/speed can be optimized, dump into production as finished product

Dropout

Dropout

- During training, some number of layer outputs are randomly ignored or "dropped out."
- the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer
- In effect, each update to a layer during training is performed with a different "view" of the configured layer.

Dropout

- Dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs.
- Makes it hard for network to overfit, it can’t focus on creating singular paths for singular inputs to the trained output, has to try and represent the pattern

Dropout

- One gain is that each training step is faster
- Generally takes longer to train as less error updating is done (some nodes are idle each execution)
- Sometimes you need bigger network than you had previously

- Often larger dropout rates earlier (in CNN think of this is that we want to ignore little tiny features earlier on)
- Often lower dropout rates later (in CNN think of this as that we've made more complex ideas, they are less likely to be overfitted)

Learning Rate

Learning Rate

- Neural networks update their weights between neuron during backpropogation
- How large this update can be is dependant on the learning rate
- A high learning rate means they update the value by a large amount, a low learning rate means a small adjustment

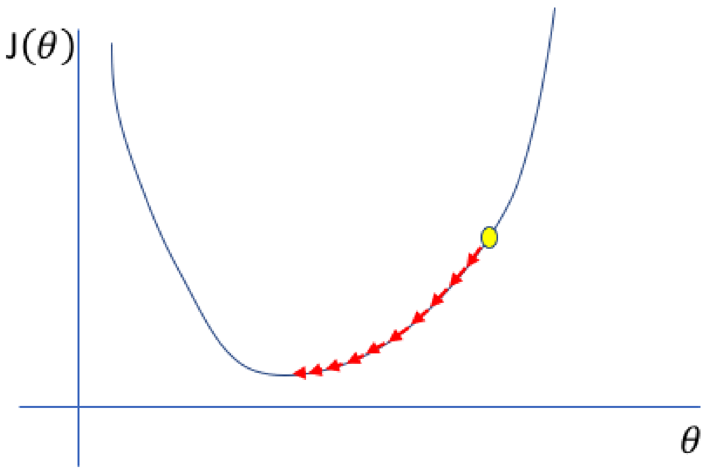
Learning Rate

$$\theta_i := \theta_i - \alpha \frac{\partial J(\theta_i)}{\partial \theta_i}$$

- Alpha is the learning rate
- J is the loss function
- You can see the derivative of the loss function/ the current weight (the activation function) being the ratio of update

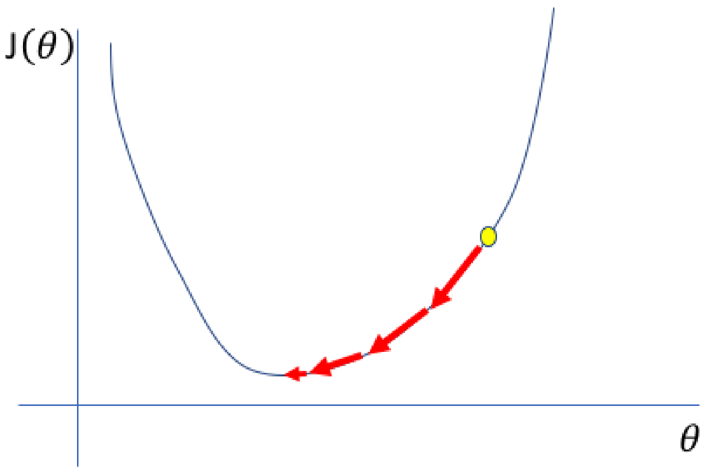
Learning Rate

Too low



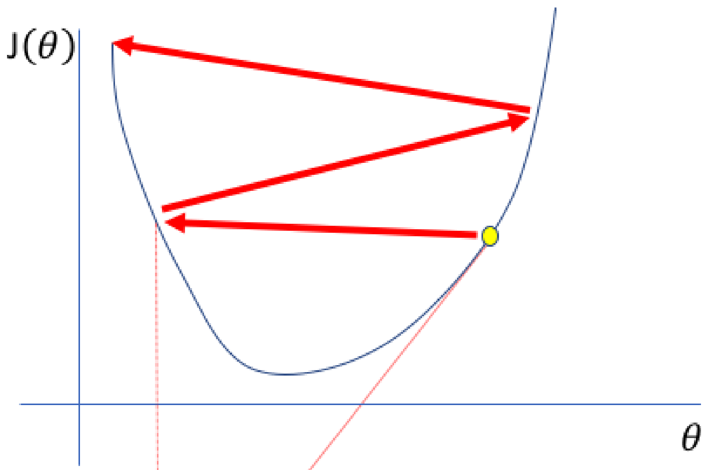
A small learning rate requires many updates before reaching the minimum point

Just right



The optimal learning rate swiftly reaches the minimum point

Too high



Too large of a learning rate causes drastic updates which lead to divergent behaviors

Learning Rate Decay

- Start with large learning rate and then reduce it over time

initial_learning_rate = 0.1

lr_schedule = **tf.keras.optimizers.schedules.ExponentialDecay**(

initial_learning_rate,

decay_steps=100000,

decay_rate=0.96,

staircase=True)

Learning Rate Decay

- Start with large learning rate and then reduce it over time

```
model.compile(
        optimizer=tf.keras.optimizers.SGD(learning_rate=lr_schedule),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])
model.fit(data, labels, epochs=5)
```

Keras Optimizers

SGD

- stochastic gradient descent, update based on learning rate multiplied by gradient (derivative ratio of loss and activation)

Adagrad

- SGD that adapts learning rates for parameters based on how often they are update

Adadelta

- robust Adagrad (adapts learning rate itself based on moving window), no need for learning rate to be set

Optimizers

Keras Optimizers

RMSprop

- maintain moving average of square of gradients, divide gradient by this square when considering an update

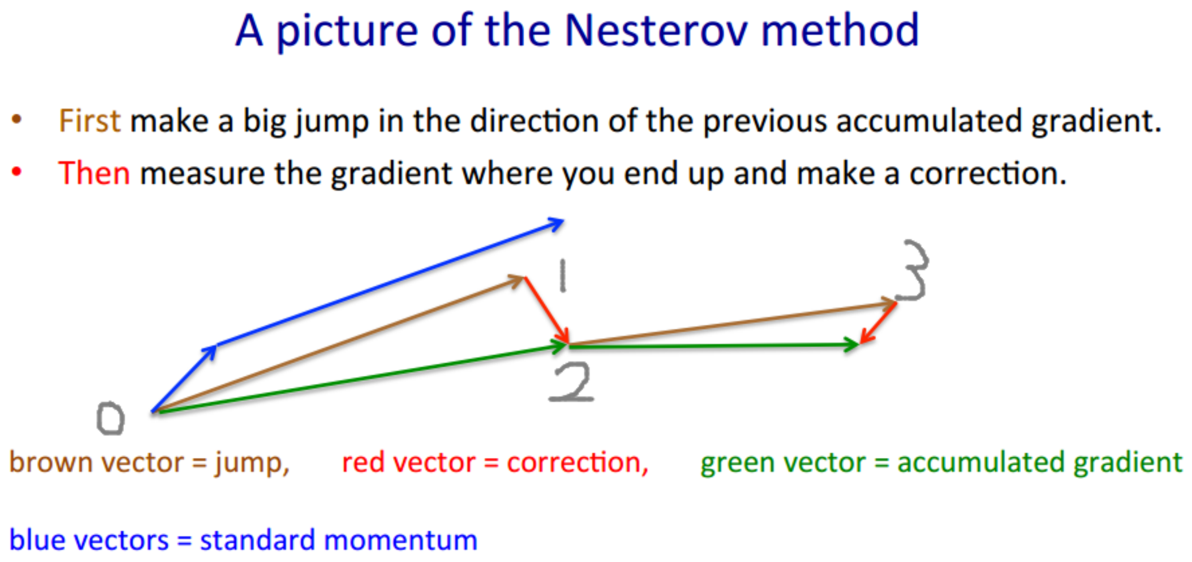
Adam

- SGD based on adaptive estimation of first and second –order moments (average and variance)
- basically RMSprop with momentum

Keras Optimizers

Adamax – Adam but based on infinity norm

Nadam – like Adam with Nesterov momentum



Ftrl – (newer a

Loss Functions

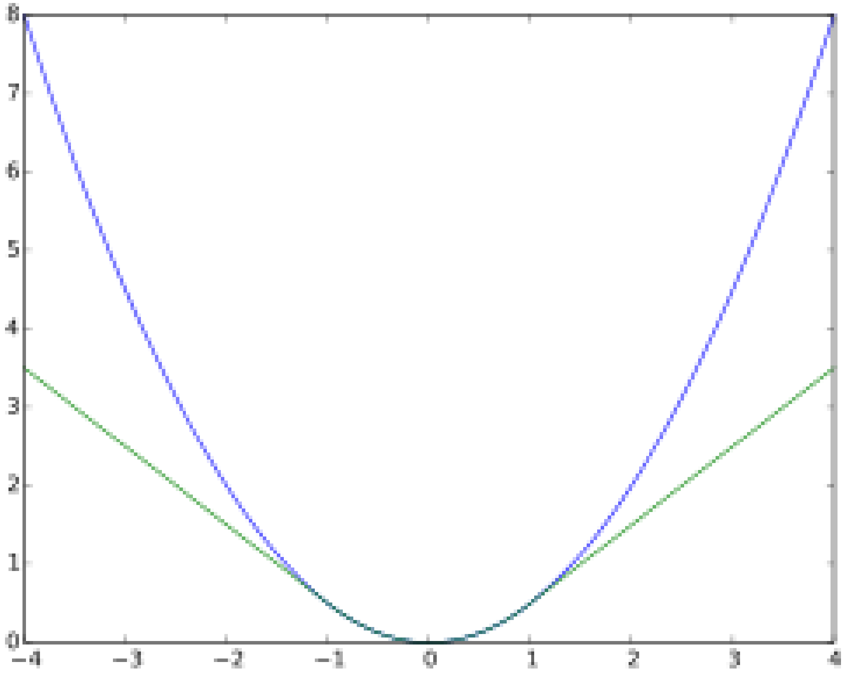
Keras loss functions – basic error

MeanSquaredError: $\left(y_{true} - y_{pred}\right)^2$

Keras loss functions – basic error

MeanSquaredError: $\left(y_{true} - y_{pred}\right)^2$

Huber: green



Keras loss functions – basic error

MeanSquaredLogarithmicError: $\left(\log(y_{true}) - \log(y_{pred})\right)^2$

MeanAbsoluteError: $|y\_true \ - \ y\_pred|$

MeanAbsolutePercentageError : $100 * \frac{|y_{true} - y_{pred}|}{y_{true}}$

Poisson: $y_{pred} - y_{true} \ * \ \log(y_{pred})$

KLDivergence (Kullback-Leibler): $y_{true} \ * \ \log(\frac{y_{true}}{y_{pred}})$

UNIVERSITY OF CALGARY

Keras loss functions (y_true and y_pred)

CosineSimilarity: cosine similarity

Hinge: $max(1 - y_{true} * y_{pred}, 0)$

- Inputs expected to be -1 or 1

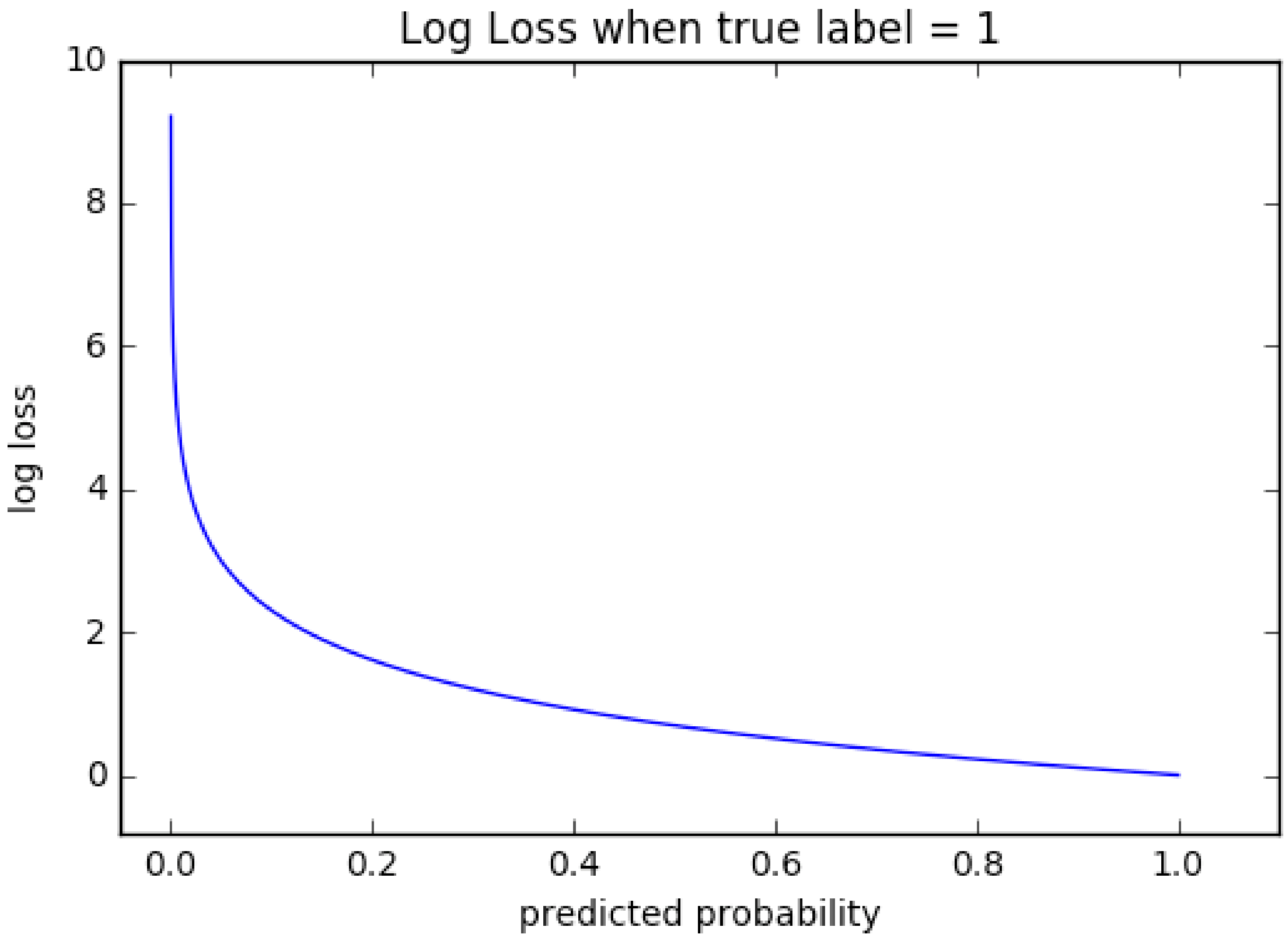
SquaredHinge: $max(1 - y_{true} * y_{pred}, 0)^2$

CategoricalHinge: $max(neg - pos + 1, 0)$

  - where $neg = sum(y_{true} * y_{pred})$ and $pos = \max(1 - y_{true})$

UNIVERSITY OF CALGARY

Keras loss functions – cross-entropy



UNIVERSITY OF CALGARY

Keras loss functions – cross-entropy

BinaryCrossentropy:

- only two label classes (0 and 1)

CategoricalCrossentropy:

- 2 or more labels in one-hot encoding 0 = [1,0,0,0] 1= [0,1,0,0] 2=[0,0,1,0], 3=[0,0,0,1]

SparseCategoricalCrossentropy:

- can use regular integer labels, 1,2,3,4

Onward to … convolutional neural networks.

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~hudsonj/

UNIVERSITY OF CALGARY