Machine Learing: Neural Networks: MNIST

CPSC 501: Advanced Programming Techniques
Winter 2025

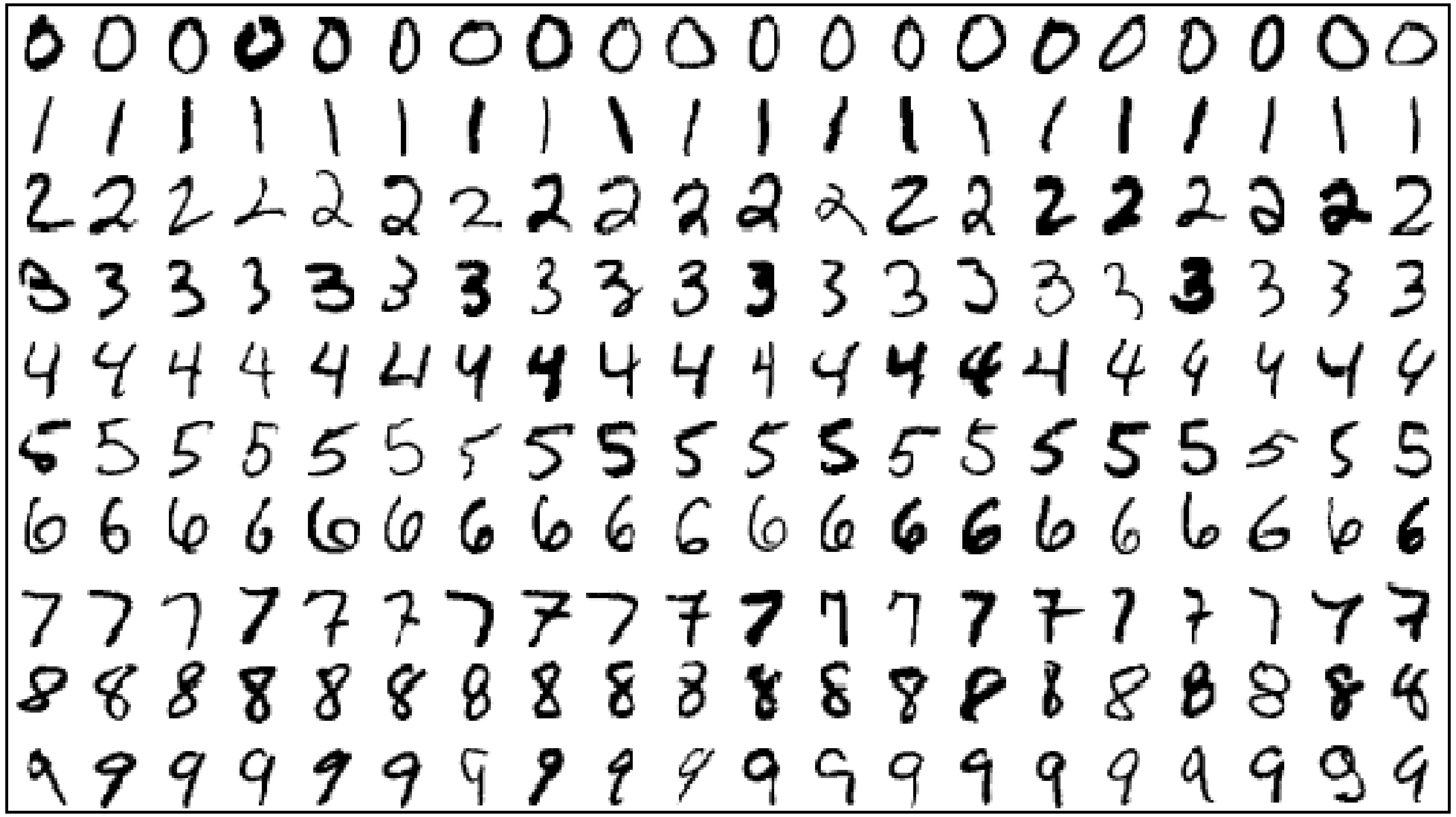
Jonathan Hudson, Ph.D
Assistant Professor (Teaching)
Department of Computer Science
University of Calgary

Friday, February 21, 2025

UNIVERSITY OF CALGARY

MNIST Database

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



UNIVERSITY OF CALGARY

Design

1. Study the problem you are trying to solve (What?)
2. Choose a model class, hyperparameters (How?)
    1. Neural networks
        1. Layers? Structure? Drop-out?
        2. Loss function: MSE? Other?
        3. Optimizer: Adam? Adam-like?
3. Prepare data (Do.)
4. Run learning algorithm to train the model (Do.)
5. Evaluate trained model (Did it work?)

Model

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

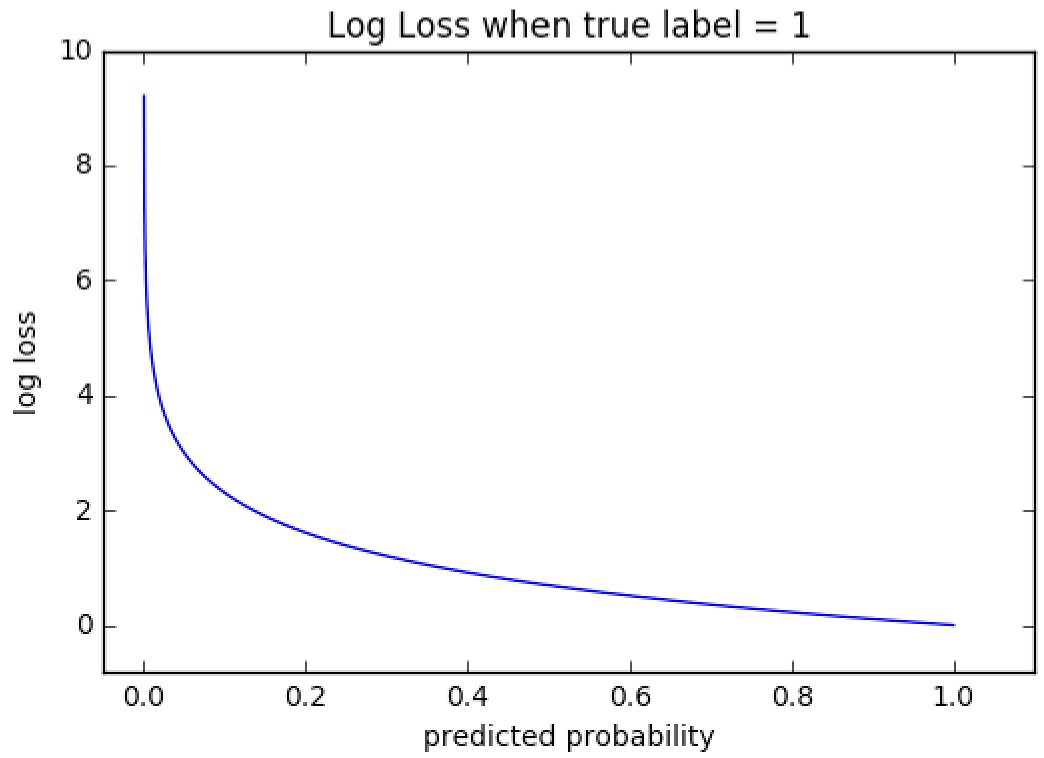
- Goal: trained model that recognizes the digit in the image

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



Variants of Cross entropy loss

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

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))
```

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

Specify loss function

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

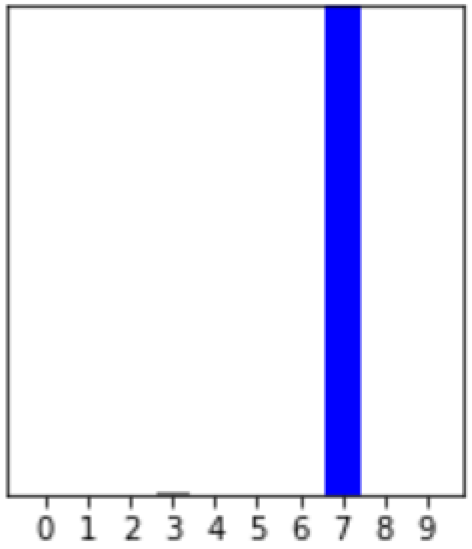
Use ‘sgd’ optimizer

We’ll discuss the loss function later in slides

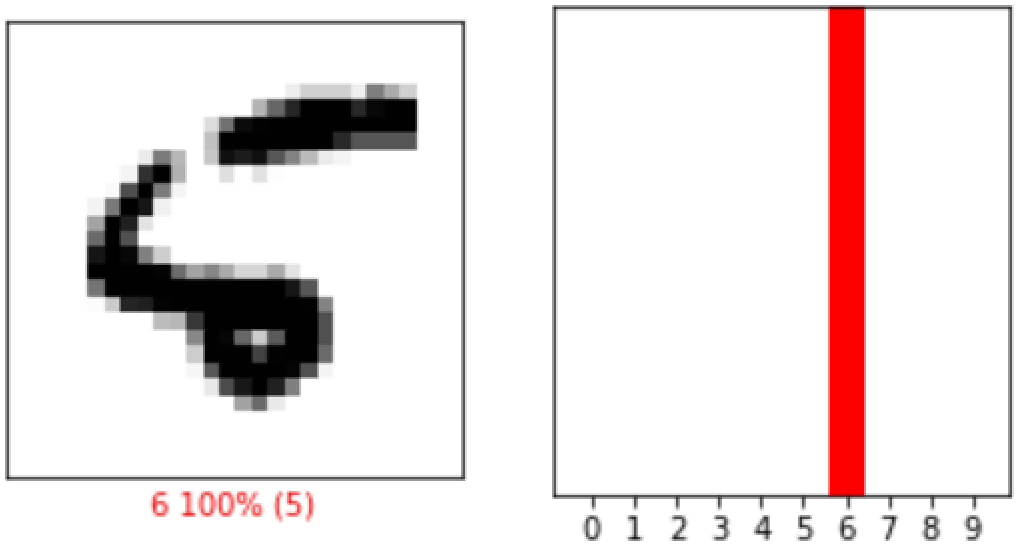
Train our model and evaluate it’s quality

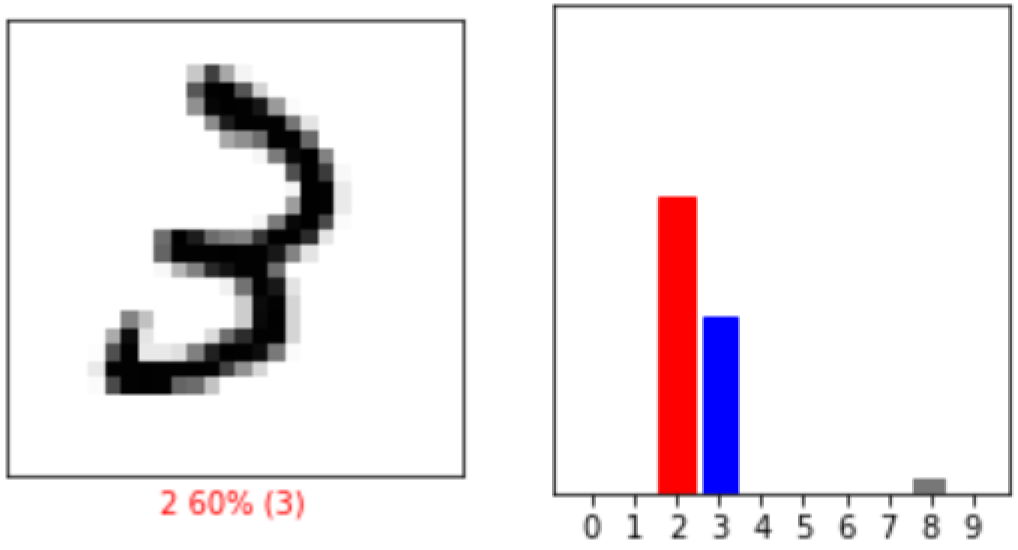
```
model.fit(x_train, y_train, epochs=5)



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}%")
```

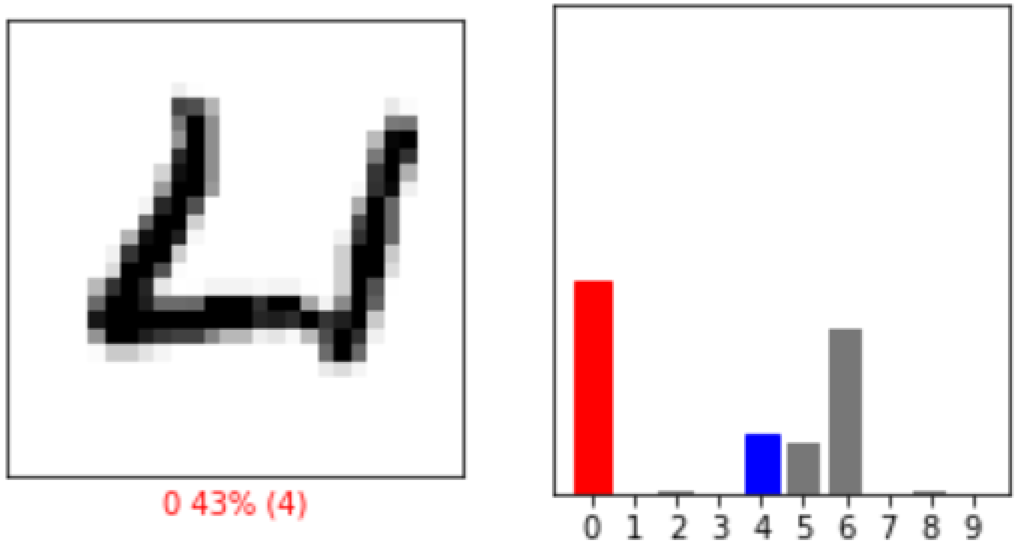
Train our model and evaluate it’s quality

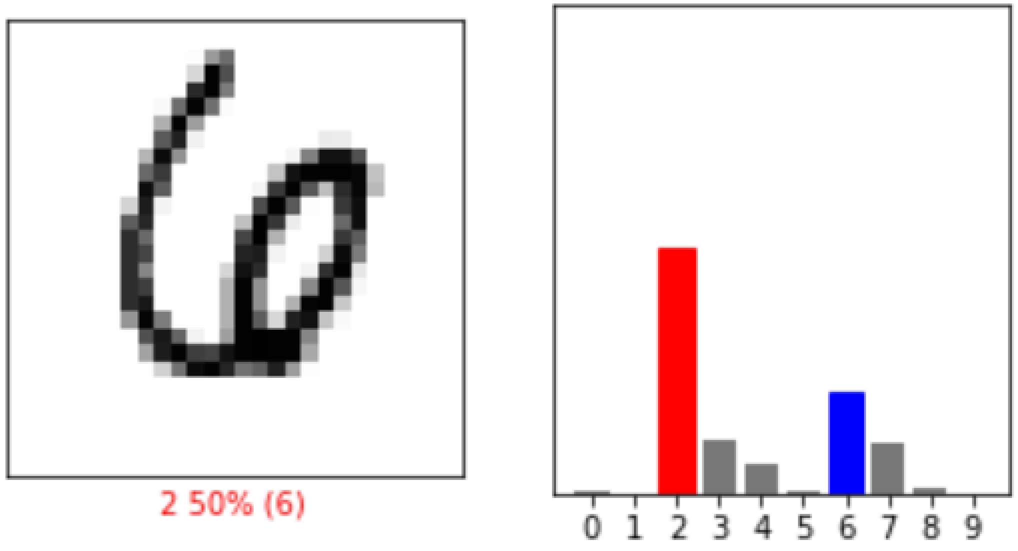


Train our model and evaluate it’s quality









Save/Load our model

model.save(‘MINST.keras’)

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

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

Splitting data?

- Why did we do this

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

- In some cases a neural network (other other model) might learn exactly what input maps to what output. Which would mean 100% performance on existing data.
- But in reality we generally want to predict things we don't have input for. 100% on known data, can sometimes result in algorithm that is much lower 66%! on new data. This is because the model didn't learn generalize patterns, but instead a mapping.
- Here we loaded a input set of data to train on, and a set to test on from MNIST.

(x_train, x_test, y_train, y_test) = train_test_split(X,y, random_state=0, train_size=0.5)

- Used to split any data into parts (here a 50% split)

Cross-validation

- If we have split data one technique to compare proposed models is cross-validation
- Split 50/50 then run two tests, each where data is input and other output, then reverse
- Often then combine the two measures to judge the model (average)
- X-fold validation is when you split data into more groups, where each sub-group takes turn as test data,
- sklearn has cross_val_score(model, X, y, cv=5) that will do this (here 5-fold)

Trade-offs

- Bias-Variance -> A more general model (like a single line of best fit, or a more varied model like a polynomial line), one might fit better but is realistically not a real model of data
- As model complexity increases it often gets easier to get a high training score, but often at a certain point the cross-validation score begins to decrease
- Larger data often helps us, more data will help a polynomial line from getting overfit as there may be enough data to keep it smoother and more realistic

Challenges

- Not enough data - easy for model to overfit and not generalize
  - “Unreasonable Effectiveness of Data” in many situations companies can often make their model better, less through design, and more through collecting more data (as often simple models are best anyway)
- Non-representative training data =- if your model has holes it will predict right over them. If you sample larger data (too little -> sample noise, too much -> sample bias)
- Poor-quality data - garbage in -> garbage out
- Irrelevant features – if you have bunch of features which are the same thing, the model will bias towards just them, can limit features, create new ones, or gather data with better features
- Overfitting/Underfitting - next

Overfit/Underfit

- An example of overfit is the polynomial model that can perfectly match data, but forgoes actually trying to be a model that data fits in
- I.e. given enough time many neural networks can learn data perfectly (especially low input quantity data)
- Underfit is when your model is too simple – this is less a problem with neural networks (unless not given enough training time due to data being too large), however an example is trying to fit non-linear data to a linear model

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

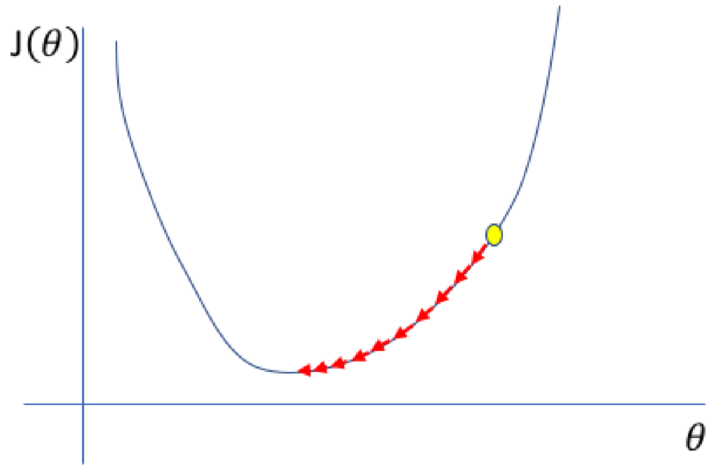
- Often larger dropout rates earlier (in CNN think of this is that we want to ignore little tiny features earlier on)
- Often lower dropout rates later (in CNN think of this as that we've made more complex ideas, they are less likely to be overfitted)

Learning Rate

- 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

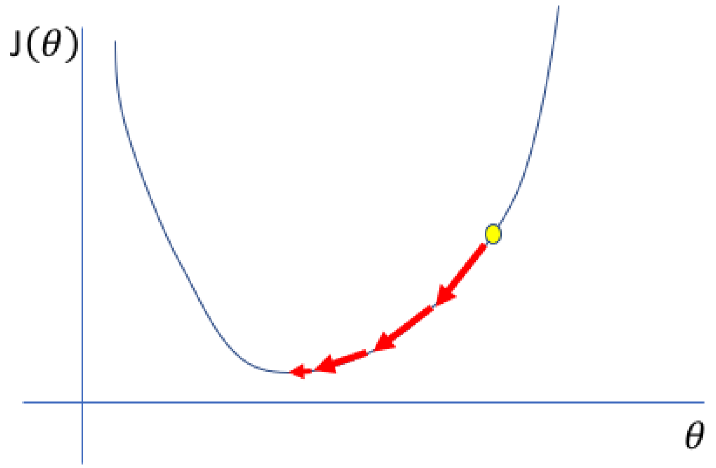
Learning Rate

Too low



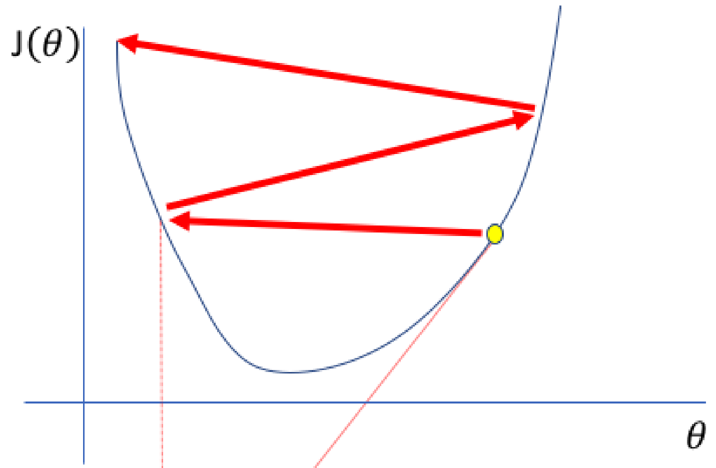
A small learning rate requires many updates before reaching the minimum point

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

- variants include Adagrad, Adadelta

RMSprop – SGD with moving average of square of gradients

Adam - RMSprop with momentum

- Variants- Adamax, Nadam, and more

Keras loss functions – predict value

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

Huber: variant of MSE

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

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

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

Many more: Poisson, KLDivergence (Kullback-Leibler), CosineSimilarity, Hinge, SquaredHinge, CategoricalHinge

Discussion

UNIVERSITY OF CALGARY

Discussion

- Decentralized knowledge representation
  ☞ possibility to parallelize (GPUs!)
- Can find pattern outside of human understanding
- Currently best way to deal with sensory data

– Network structure determines what can be learned
  ☞ must be provided by user

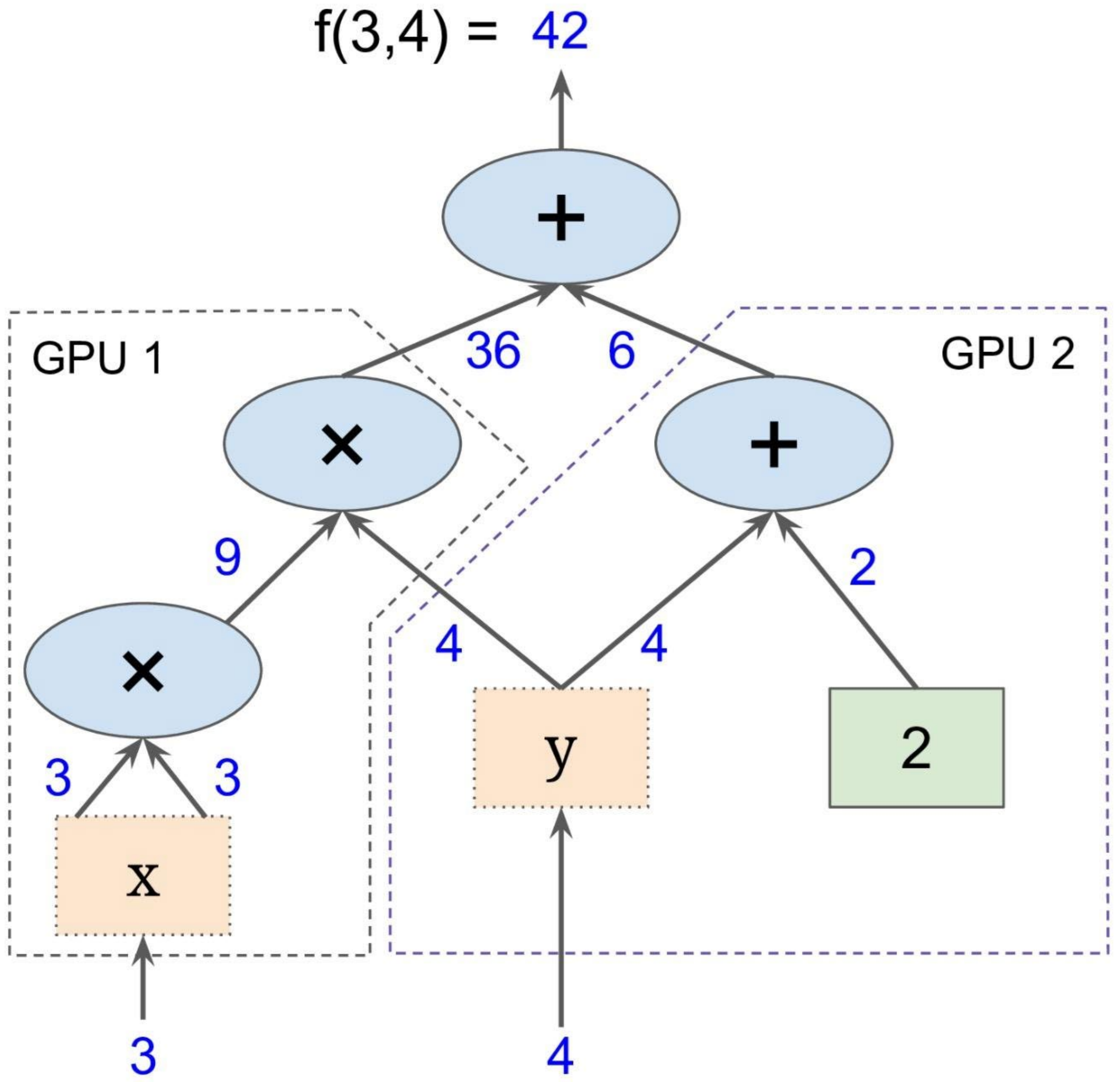
– Represented knowledge not understandable by humans

– Learning can take very long

– Too many learning procedures: when to choose which?

NN Bonus! -> Subgraphs Let us use Compute Units

Possible to break graphs into several chunks and run them parallelly across multiple CPUs, GPUs, TPUs, or other devices



Graph from *Hands-On Machine Learning with Scikit-Learn and TensorFlow*

Why graphs

1. Save computation. Only run subgraphs that lead to the values you want to fetch.
2. Break computation into small, differential pieces to facilitate auto-differentiation
3. Facilitate distributed computation, spread the work across multiple CPUs, GPUs, TPUs, or other devices
4. Many common machine learning models are taught and visualized as directed graphs

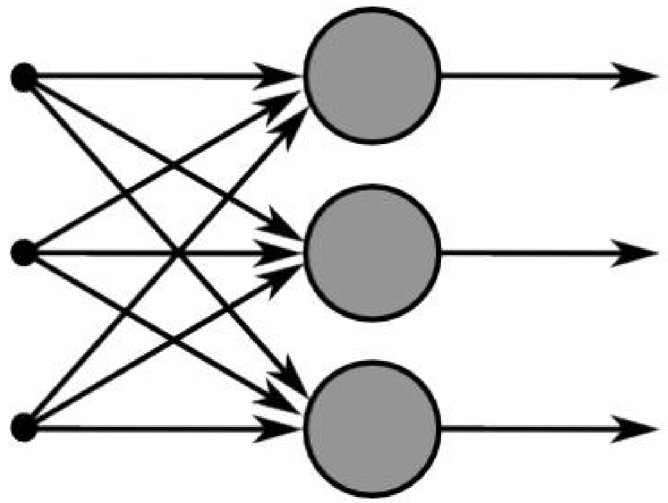


Figure 3: This image captures how multiple sigmoid units are stacked on the right, all of which receive the same input x.

Onward to … Convolutional Neural Networks

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

UNIVERSITY OF CALGARY