DS 4400

Machine Learning and Data Mining I

Alina Oprea
Associate Professor, CCIS
Northeastern University

November 8 2018

Logistics

- HW 3 is due on Tuesday, November 13
- Project Milestone is due on Tuesday, November 20
- Class on November 27 is cancelled
- Final project presentations
  - Monday, December 3, 3-5:30pm in ISEC 655
- Final exam
  - Tuesday, Dec 11, 2-5pm in ISEC 655

Review

- To train neural networks, need to decide first on architecture
  - Number of layers, number of hidden units, connections between neurons, activation functions
- Randomly initialize parameters
- For each training example, use forward propagation to compute prediction
- Use backpropagation to propagate the error from last layer back into the network

References

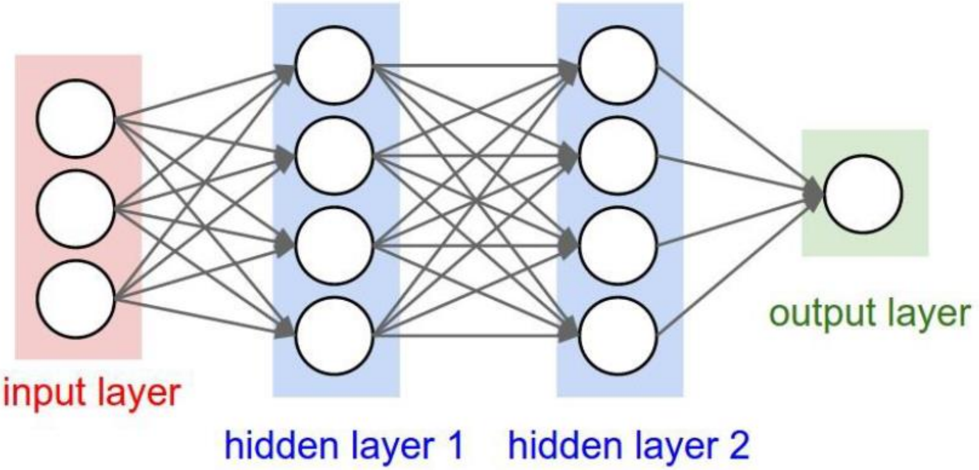
- Stanford tutorial on training Multi-Layer Neural Networks
  - http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/
- Notes on backpropagation by Andrew Ng
  - http://cs229.stanford.edu/notes/cs229-notes-backprop.pdf
- Deep learning notes by Andrew Ng
  - http://cs229.stanford.edu/notes/cs229-notes-deep_learning.pdf

Outline

- Training with backpropagation
  - Gradient-Descent Algorithm
  - Derivation of gradients
  - Stochastic and Mini-Batch Gradient Descent
- Lab
  - MNIST dataset
- Regularization for Neural Networks
  - L2/L1 regularization
  - Dropout

Forward Propagation

- The input neurons first receive the data features of the object. After processing the data, they send their output to the first hidden layer.
- The hidden layer processes this output and sends the results to the next hidden layer.
- This continues until the data reaches the final output layer, where the output value determines the object's classification.
- This entire process is known as Forward Propagation, or Forward prop.



Learning in NN: Backpropagation

- Similar to the perceptron learning algorithm, we cycle through our examples
  - If the output of the network is correct, no changes are made
  - If there is an error, weights are adjusted to reduce the error

- The trick is to assess the blame for the error and divide it among the contributing weights

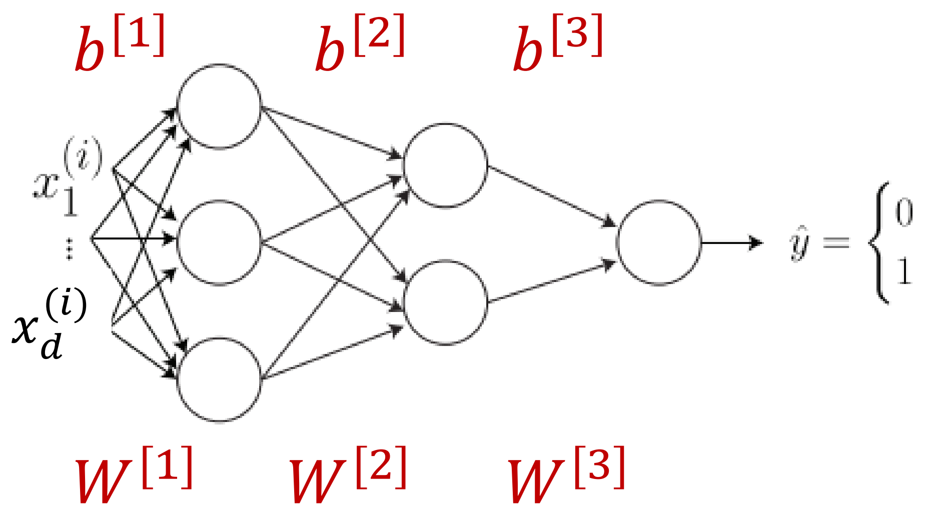
Error at last layer can be measured, but it is challenging to determine error at intermediate hidden layers

GD for Neural Networks

- Initialization
  - For all layers ℓ
    - Set $W^{[\ell]}, b^{[\ell]}$ at random
- Backpropagation
  - Fix learning rate α
  - For all layers ℓ (starting backwards)
    - $W^{[\ell]} = W^{[\ell]} - \alpha \sum_{i=1}^{N} \frac{\partial L(\hat{y}^{(i)}, y^{(i)})}{\partial W^{[\ell]}}$
    - $b^{[\ell]} = b^{[\ell]} - \alpha \sum_{i=1}^{N} \frac{\partial L(\hat{y}^{(i)}, y^{(i)})}{\partial b^{[\ell]}}$

Example

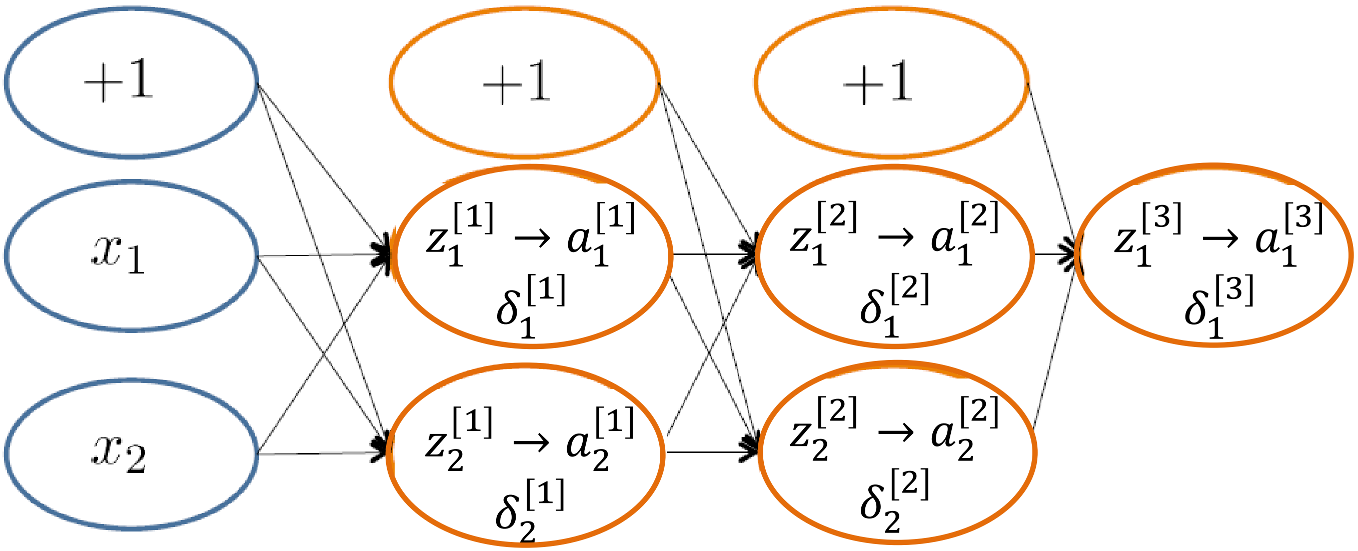
Training data
Dimension d



$$z^{[1]} = W^{[1]}x^{(i)} + b^{[1]}$$
$$a^{[1]} = g(z^{[1]})$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = g(z^{[2]})$$
$$z^{[3]} = W^{[3]}a^{[2]} + b^{[3]}$$
$$\hat{y}^{(i)} = a^{[3]} = g(z^{[3]})$$

Parameters are initialized with random values (not all 0)!

Backpropagation Intuition

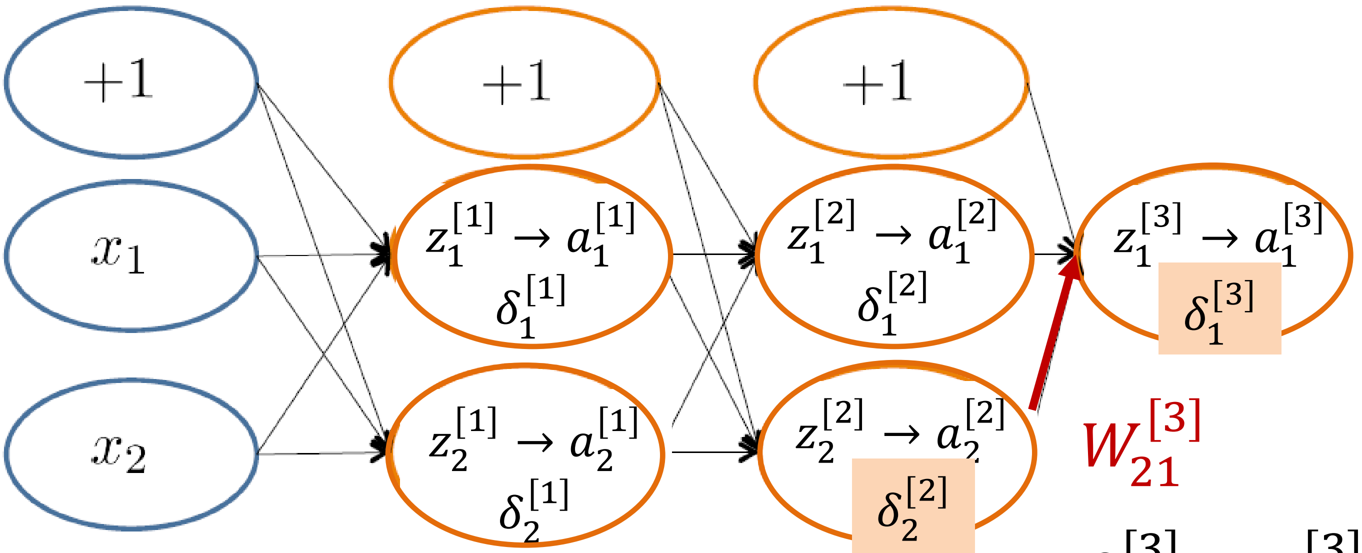


$\delta_j^{(l)}$ = "error" of node j in layer l

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}\,(x^{(i)})$

$$cost\left(x^{(i)}\right) = y^{(i)} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \log(1 - h_\theta(x^{(i)}))$$

Backpropagation Intuition



$$\delta_1^{[3]} \approx a_1^{[3]} - y$$

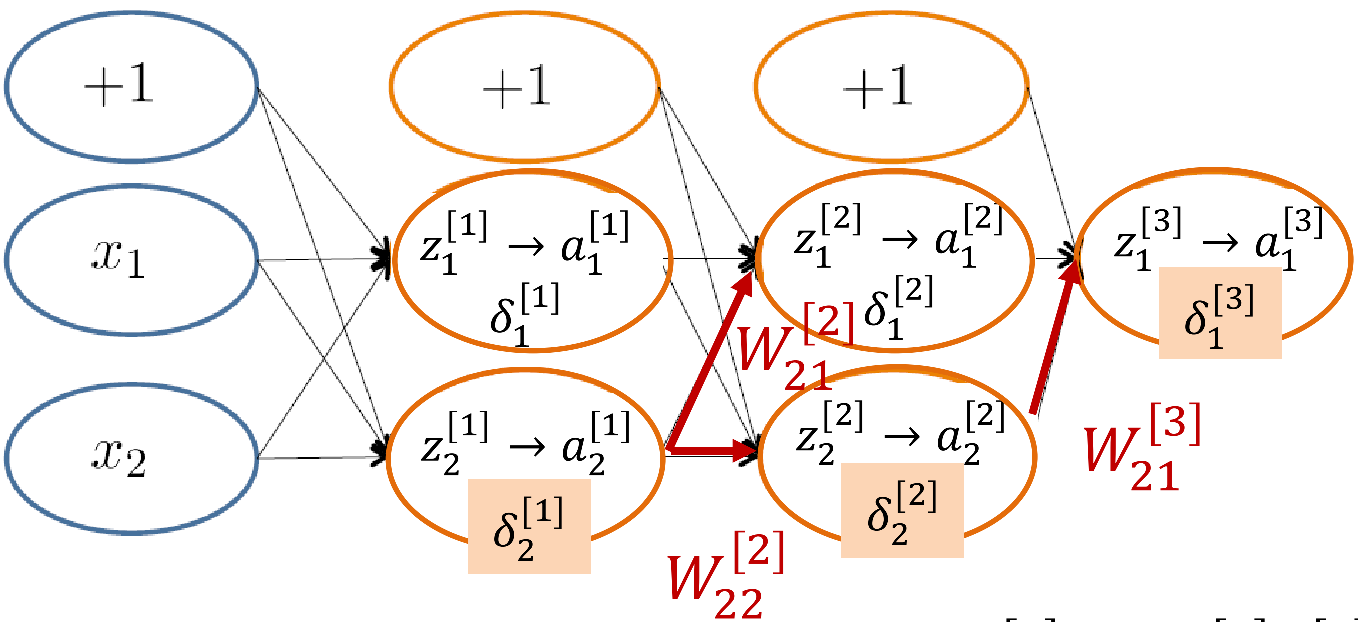
$$\delta_2^{[2]} \approx \delta_1^{[3]} W_{21}^{[3]}$$

$\delta_j^{(l)}$ = "error" of node j in layer l

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}\ (x^{(i)})$

$$cost\left(x^{(i)}\right) = y^{(i)} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \log(1 - h_\theta(x^{(i)}))$$

Backpropagation Intuition



$$\delta_2^{[1]} \approx W_{21}^{[2]}\delta_1^{[2]} + W_{22}^{[2]}\delta_2^{[2]}$$

$\delta_j^{(l)}$ = "error" of node j in layer l

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}\,(x^{(i)})$

$$cost\left(x^{(i)}\right) = y^{(i)} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right)\log(1 - h_\theta(x^{(i)}))$$

Training

- Training data $x^{(1)}, \mathrm{y}^{(1)}, \dots x^{(N)}, \mathrm{y}^{(\mathrm{N})}$
- One training example $x^{(i)} = \left(x_1^{(i)}, \dots x_d^{(i)}\right)$, label $y^{(i)}$
- One forward pass through the network
  - Compute prediction $\hat{y}^{(i)}$
- Loss function for one example
  - $L(\hat{y}, y) = -[(1 - y)\log(1 - \hat{y}) + y \log \hat{y}]$

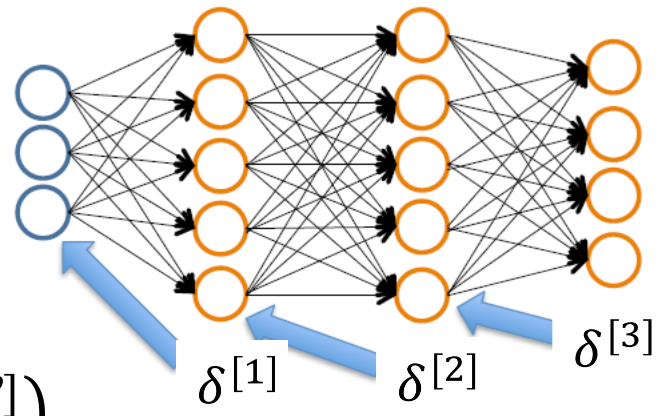
Cross-entropy loss

- Loss function for training data
  - $J(W, b) = \frac{1}{N} \sum_i L\left(\hat{y}^{(i)}, y^{(i)}\right) + \lambda R(W, b)$

Backpropagation

Let $\delta_j^{(l)}$ = "error" of node j in layer l

$$L(y, \hat{y}) = -[(1-y)\log(1-\hat{y}) + \text{y}\log\hat{y}]$$



- Definitions
  - $z^{[\ell]} = W^{[\ell]} a^{[\ell-1]} + b^{[\ell]}, a^{[\ell]} = g\left(z^{[\ell]}\right)$
  - $\delta^{[\ell]} = \frac{\partial L(\hat{y}, y)}{\partial z^{[\ell]}}$
-
-

Example: Last Layer (3)

- $\delta^{[3]} = \frac{\partial L(\hat{y},y)}{\partial z^{[3]}} = \frac{\partial L(\hat{y},y)}{\partial \hat{y}} g'\left(z^{[3]}\right); \hat{y} = g\left(z^{[3]}\right) = a^{[3]}$
- $\frac{\partial L(\hat{y},y)}{\partial \hat{y}} = - \frac{\partial[(1-y)\log(1-\hat{y})+y log\ \hat{y}]}{\partial \hat{y}} = \frac{1-y}{1-\hat{y}} - \frac{y}{\hat{y}} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}$
- $\delta^{[3]} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} g'\left(z^{[3]}\right)$

  $= \frac{a^{[3]}-y}{g\left(z^{[3]}\right)\left(1-g\left(z^{[3]}\right)\right)} \mathrm{g}\left(\mathrm{z}^{[3]}\right)\left(1-\mathrm{g}\left(\mathrm{z}^{[3]}\right)\right) = \mathrm{a}^{[3]} - \mathrm{y}$
- $\frac{\partial L(\hat{y},y)}{\partial W^{[3]}} = \delta^{[3]} a^{[2]T} = \left(a^{[3]} - y\right) a^{[2]T}$
- $\frac{\partial L(\hat{y},y)}{\partial b^{[3]}} = \mathrm{a}^{[3]} - \mathrm{y}$

$$g(x) = \sigma(x) = \frac{1}{1+e^{-x}}$$
$$g'(x) = \sigma'(x) = \sigma(x)(1-\sigma(x))$$

Example: Layer 2

- $\delta^{[2]} = \frac{\partial L(\hat{y},y)}{\partial z^{[2]}} = \delta^{[3]} W^{[3]} g'(z^{[2]})$

- $\frac{\partial L(\hat{y},y)}{\partial\, W^{[2]}} = \delta^{[2]} a^{[1]T} = \delta^{[3]} W^{[3]} g'(z^{[2]})\, a^{[1]T} =$

  $= [a^{[3]} - y] W^{[3]} g(z^{[2]})\, (1 - g(z^{[2]})) a^{[1]T}$

- $\frac{\partial L(\hat{y},y)}{\partial\, b^{[2]}} = [a^{[3]} - y] W^{[3]} g(z^{[2]})\, (1 - g(z^{[2]}))$

$$g(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$
$$g'(x) = \sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Batch Perceptron

Given training data $\left\{(\boldsymbol{x}^{(i)}, y^{(i)})\right\}_{i=1}^{n}$
Let $\boldsymbol{\theta} \leftarrow [0, 0, \ldots, 0]$
Repeat:
    Let $\boldsymbol{\Delta} \leftarrow [0, 0, \ldots, 0]$
    for $i = 1 \ldots n$, do
        if $y^{(i)} \boldsymbol{x}^{(i)} \boldsymbol{\theta} \leq 0$ // prediction for ith instance is incorrect
            $\boldsymbol{\Delta} \leftarrow \boldsymbol{\Delta} + y^{(i)} \boldsymbol{x}^{(i)}$
    $\boldsymbol{\Delta} \leftarrow \boldsymbol{\Delta} / n$ // compute average update
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{\Delta}$
Until $\|\boldsymbol{\Delta}\|_2 < \epsilon$

EPOCH

- Simplest case: $\alpha = 1$ and don't normalize, yields the fixed increment perceptron
- Each increment of outer loop is called an **epoch**

Backpropagation

Set $\Delta_{ij}^{(l)} = 0 \quad \forall l, i, j$ (Used to accumulate gradient)

For each training instance $(x^{(i)}, y^{(i)})$

Set $\mathbf{a}^{(1)} = \mathbf{x}_i$

Compute $\{\mathbf{a}^{(2)}, \ldots, \mathbf{a}^{(L)}\}$ via forward propagation

Compute $\boldsymbol{\delta}^{(L)} = \mathbf{a}^{(L)} - y^{(i)}$

Compute errors $\{\boldsymbol{\delta}^{(L-1)}, \ldots, \boldsymbol{\delta}^{(2)}\}$

Compute gradients $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

Average gradient is $\frac{\Delta_{ij}^{[\ell]}}{N}$

Training NN with Backpropagation

Given training set $(x_1, y_1), \ldots, (x_N, y_N)$
Initialize all parameters $W^{[\ell]}, b^{[\ell]}$ randomly, for all layers ℓ
Loop

Set $\Delta_{ij}^{(l)} = 0 \quad \forall l, i, j$ (Used to accumulate gradient)
For each training instance $(x^{(i)}, y^{(i)})$
  Set $\mathbf{a}^{(1)} = \mathbf{x}_i$
  Compute $\{\mathbf{a}^{(2)}, \ldots, \mathbf{a}^{(L)}\}$ via forward propagation
  Compute $\boldsymbol{\delta}^{(L)} = \mathbf{a}^{(L)} - y^{(i)}$
  Compute errors $\{\boldsymbol{\delta}^{(L-1)}, \ldots, \boldsymbol{\delta}^{(2)}\}$
  Compute gradients $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

EPOCH

Update weights via gradient step

- $W_{ij}^{[\ell]} = W_{ij}^{[\ell]} - \alpha \frac{\Delta_{ij}^{[\ell]}}{N}$
- Similar for $b_{ij}^{[\ell]}$

Until weights converge or maximum number of epochs is reached

GD for Neural Networks

- Initialization
  - For all layers ℓ
    - Set $W^{[\ell]}, b^{[\ell]}$ at random
- Backpropagation
  - Fix learning rate α
  - For all layers ℓ (starting backwards)
    - $W^{[\ell]} = W^{[\ell]} - \alpha \sum_{i=1}^{N} \frac{\partial L(\hat{y}^{(i)}, y^{(i)})}{\partial W^{[\ell]}}$
    - $b^{[\ell]} = b^{[\ell]} - \alpha \sum_{i=1}^{N} \frac{\partial L(\hat{y}^{(i)}, y^{(i)})}{\partial b^{[\ell]}}$

This is expensive!

Stochastic Gradient Descent (SGD)

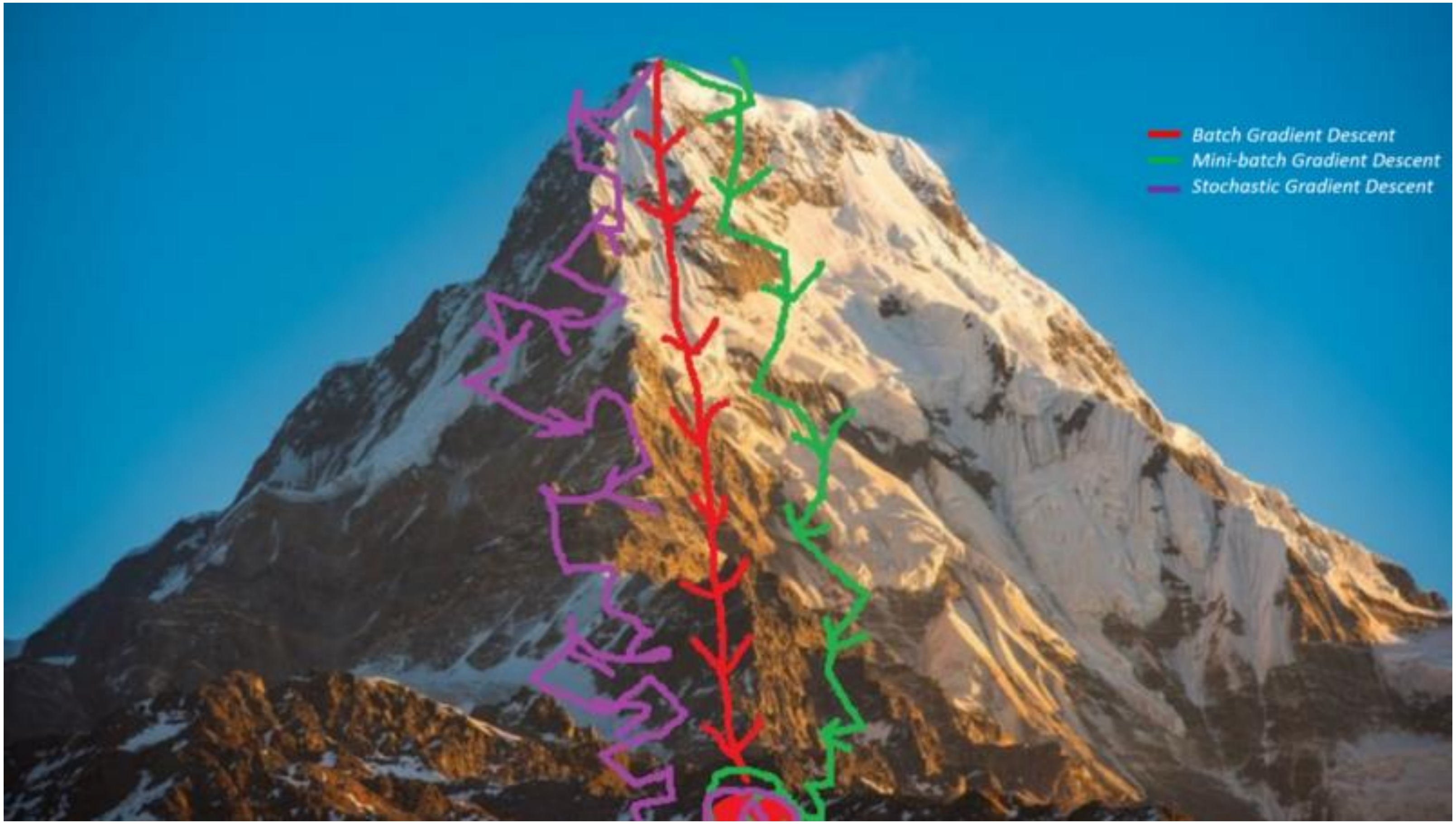
- Initialization
  - For all layers ℓ
    - Set $W^{[\ell]}, b^{[\ell]}$ at random
- Backpropagation
  - Fix learning rate α
  - For all layers ℓ (starting backwards)
    - For all training examples $x^{(i)}, y^{(i)}$
      - $W^{[\ell]} = W^{[\ell]} - \alpha \frac{\partial L(\hat{y}^{(i)}, y^{(i)})}{\partial W^{[\ell]}}$
      - $b^{[\ell]} = b^{[\ell]} - \alpha \frac{\partial L(\hat{y}^{(i)}, y^{(i)})}{\partial b^{[\ell]}}$

Incremental version of GD

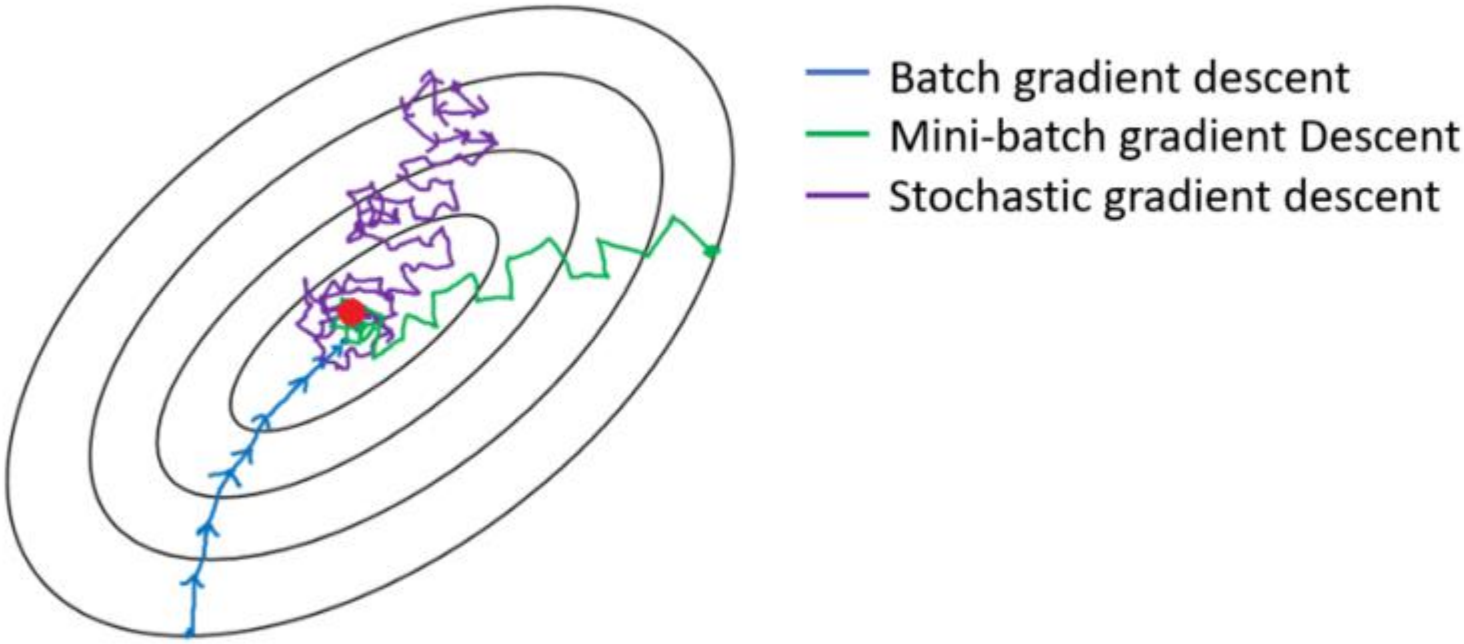
Mini-batch Gradient Descent

- Initialization
  - For all layers ℓ
    - Set $W^{[\ell]}, b^{[\ell]}$ at random
- Backpropagation
  - Fix learning rate α
  - For all layers ℓ (starting backwards)
    - For all batches b of size B with training examples $x^{(ib)}, y^{(ib)}$
      - $W^{[\ell]} = W^{[\ell]} - \alpha \sum_{i=1}^{B} \frac{\partial L(\hat{y}^{(ib)}, y^{(ib)})}{\partial W^{[\ell]}}$
      - $b^{[\ell]} = b^{[\ell]} - \alpha \sum_{i=1}^{B} \frac{\partial L(\hat{y}^{(ib)}, y^{(ib)})}{\partial b^{[\ell]}}$

Gradient Descent Variants

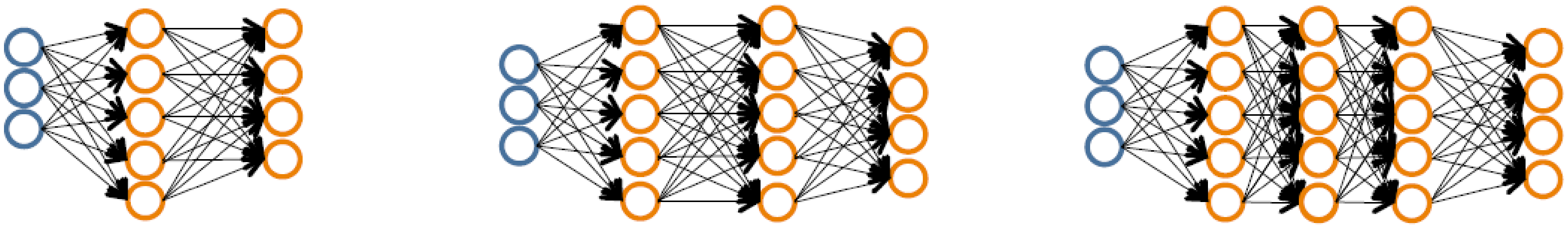


Gradient Descent Variants



Training Neural Networks

Pick a network architecture (connectivity pattern between nodes)



- # input units = # of features in dataset
- # output units = # classes

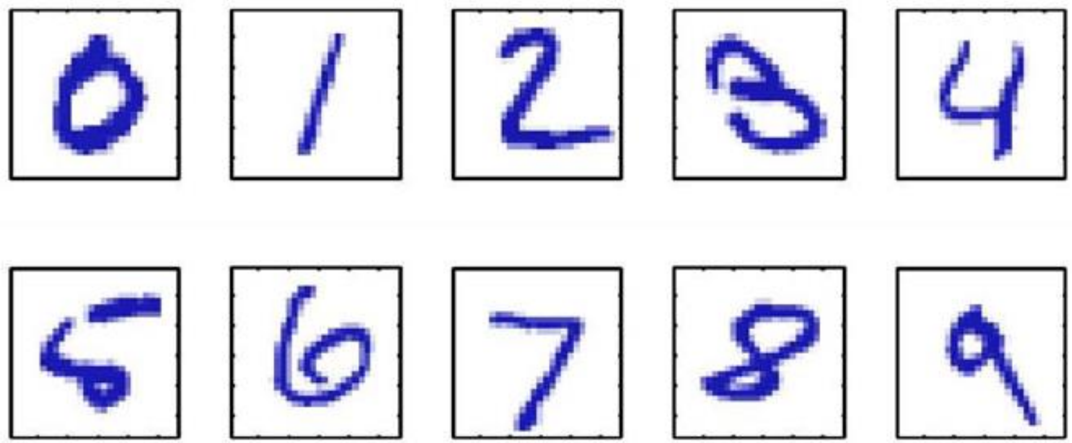
Reasonable default: 1 hidden layer

- or if >1 hidden layer, have same # hidden units in every layer (usually the more the better)

Training Neural Networks

- Randomly initialize weights
- Implement forward propagation to get prediction $\widehat{y}_i$ for any training instance x_i
- Compute loss function $L(\hat{y}_i, y_i)$
- Implement backpropagation to compute partial derivatives $\frac{\partial L(\hat{y}^{(i)}, y^{(i)})}{\partial W^{[\ell]}}$ and $\frac{\partial L(\hat{y}^{(i)}, y^{(i)})}{\partial b^{[\ell]}}$
- Use gradient descent with backpropagation to compute parameter values that optimize loss

MNIST: Handwritten digit recognition



Images are 28 x 28 pixels

Represent input image as a vector $\mathbf{x} \in \mathbb{R}^{784}$

Learn a classifier $f(\mathbf{x})$ such that,

$$f : \mathbf{x} \to \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Predict the digit

Multi-class classifier

Lab – Feed Forward NN

```
import time
import numpy as np
from keras.utils import np_utils
import keras.callbacks as cb
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import RMSprop
from keras.datasets import mnist

import matplotlib
matplotlib.use('agg')
import matplotlib.pyplot as plt
```

Import modules

```
def load_data():
    print("Loading data")
    (X_train, y_train), (X_test, y_test) = mnist.load_data()

    X_train = X_train.astype('float32')
    X_test = X_test.astype('float32')

    # Normalize
    X_train /= 255
    X_test /= 255

    y_train = np_utils.to_categorical(y_train, 10)
    y_test = np_utils.to_categorical(y_test, 10)

    X_train = np.reshape(X_train, (60000, 784))
    X_test = np.reshape(X_test, (10000, 784))

    print("Data Loaded")
    return [X_train, X_test, y_train, y_test]
```

Load MNIST data

Define NN architecture

```
def init_model():
    start_time = time.time()

    print("Compiling Model")
    model = Sequential()
    model.add(Dense(500, input_dim=784))
    model.add(Activation('relu'))
    model.add(Dropout(0.4))
    model.add(Dense(300))
    model.add(Activation('relu'))
    model.add(Dropout(0.4))
    model.add(Dense(10))
    model.add(Activation('softmax'))

    rms = RMSprop()
    model.compile(loss='categorical_crossentropy', optimizer=rms, metrics=['accuracy'])

    print("Model finished"+format(time.time() - start_time))
    return model
```

500 hidden units

ReLU activation

Dropout regularization

Softmax activation

Loss function

Optimizer

Train and evaluate

```
def run_network(data=None, model=None, epochs=20, batch=256):
    try:
        start_time = time.time()
        if data is None:
            X_train, X_test, y_train, y_test = load_data()
        else:
            X_train, X_test, y_train, y_test = data

        if model is None:
            model = init_model()

        history = LossHistory()

        print("Training model")
        model.fit(X_train, y_train, nb_epoch=epochs, batch_size=batch,
                  callbacks=[history],
                  validation_data=(X_test, y_test), verbose=2)

        print("Training duration:"+format(time.time() - start_time))
        score = model.evaluate(X_test, y_test, batch_size=16)

        print("\nNetwork's test loss and accuracy:"+format(score))
        return model, history.losses
```

```
class LossHistory(cb.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []

    def on_batch_end(self, batch, logs={}):
        batch_loss = logs.get('loss')
        self.losses.append(batch_loss)
```

Run code

```
model, losses = run_network()
```

```
[alina@dome MNIST]$ python3 ffnn.py
Using TensorFlow backend.
Loading data
Data loaded
Compiling Model
WARNING:tensorflow:From /usr/local/lib/python3.6/site-packages/keras/backend/ten
sorflow_backend.py:2755: calling reduce_sum (from tensorflow.python.ops.math_ops
) with keep_dims is deprecated and will be removed in a future version.
Instructions for updating:
keep_dims is deprecated, use keepdims instead
WARNING:tensorflow:From /usr/local/lib/python3.6/site-packages/keras/backend/ten
sorflow_backend.py:1290: calling reduce_mean (from tensorflow.python.ops.math_op
s) with keep_dims is deprecated and will be removed in a future version.
Instructions for updating:
keep_dims is deprecated, use keepdims instead
Model finished0.22017550468444824
Training model
/usr/local/lib/python3.6/site-packages/keras/models.py:848: UserWarning: The `nb
_epoch` argument in `fit` has been renamed `epochs`.
  warnings.warn('The `nb_epoch` argument in `fit` '
Train on 60000 samples, validate on 10000 samples
```

```
Epoch 1/20
2018-08-29 20:02:45.664799: I tensorflow/core/common_runtime/gpu/gpu_device.c
name: TITAN X (Pascal) major: 6 minor: 1 memoryClockRate(GHz): 1.531
pciBusID: 0000:83:00.0
totalMemory: 11.90GiB freeMemory: 406.00MiB
2018-08-29 20:02:45.664916: I tensorflow/core/common_runtime/gpu/gpu_device.c
2018-08-29 20:02:46.044142: I tensorflow/core/common_runtime/gpu/gpu_device.c
lica:0/task:0/device:GPU:0 with 127 MB memory) -> physical GPU (device: 0, na
te capability: 6.1)
2s - loss: 0.3561 - acc: 0.8916 - val_loss: 0.1441 - val_acc: 0.9550
Epoch 2/20
1s - loss: 0.1545 - acc: 0.9538 - val_loss: 0.0939 - val_acc: 0.9706
Epoch 3/20
1s - loss: 0.1128 - acc: 0.9663 - val_loss: 0.0796 - val_acc: 0.9744
Epoch 4/20
1s - loss: 0.0923 - acc: 0.9718 - val_loss: 0.0739 - val_acc: 0.9767
Epoch 5/20
1s - loss: 0.0803 - acc: 0.9755 - val_loss: 0.0792 - val_acc: 0.9767
Epoch 6/20
1s - loss: 0.0722 - acc: 0.9782 - val_loss: 0.0701 - val_acc: 0.9800
Epoch 7/20
1s - loss: 0.0645 - acc: 0.9802 - val_loss: 0.0720 - val_acc: 0.9814
Epoch 8/20
1s - loss: 0.0590 - acc: 0.9824 - val_loss: 0.0700 - val_acc: 0.9811
Epoch 9/20
1s - loss: 0.0522 - acc: 0.9833 - val_loss: 0.0723 - val_acc: 0.9792
Epoch 10/20
1s - loss: 0.0522 - acc: 0.9844 - val_loss: 0.0659 - val_acc: 0.9826
Epoch 11/20
1s - loss: 0.0460 - acc: 0.9861 - val_loss: 0.0646 - val_acc: 0.9847
Epoch 12/20
1s - loss: 0.0442 - acc: 0.9867 - val_loss: 0.0696 - val_acc: 0.9825
Epoch 13/20
1s - loss: 0.0438 - acc: 0.9866 - val_loss: 0.0731 - val_acc: 0.9824
Epoch 14/20
1s - loss: 0.0396 - acc: 0.9881 - val_loss: 0.0702 - val_acc: 0.9837
Epoch 15/20
1s - loss: 0.0371 - acc: 0.9888 - val_loss: 0.0822 - val_acc: 0.9831
Epoch 16/20
1s - loss: 0.0372 - acc: 0.9887 - val_loss: 0.0736 - val_acc: 0.9837
Epoch 17/20
1s - loss: 0.0347 - acc: 0.9893 - val_loss: 0.0755 - val_acc: 0.9830
Epoch 18/20
1s - loss: 0.0323 - acc: 0.9901 - val_loss: 0.0749 - val_acc: 0.9839
Epoch 19/20
1s - loss: 0.0340 - acc: 0.9899 - val_loss: 0.0754 - val_acc: 0.9831
Epoch 20/20
1s - loss: 0.0326 - acc: 0.9907 - val_loss: 0.0839 - val_acc: 0.9815
Training duration:25.691107034683228
 9568/10000 [===========================>..] - ETA: 0s
Network's test loss and accuracy:[0.083858822271390659, 0.98150000000000004]
```

Epoch Output

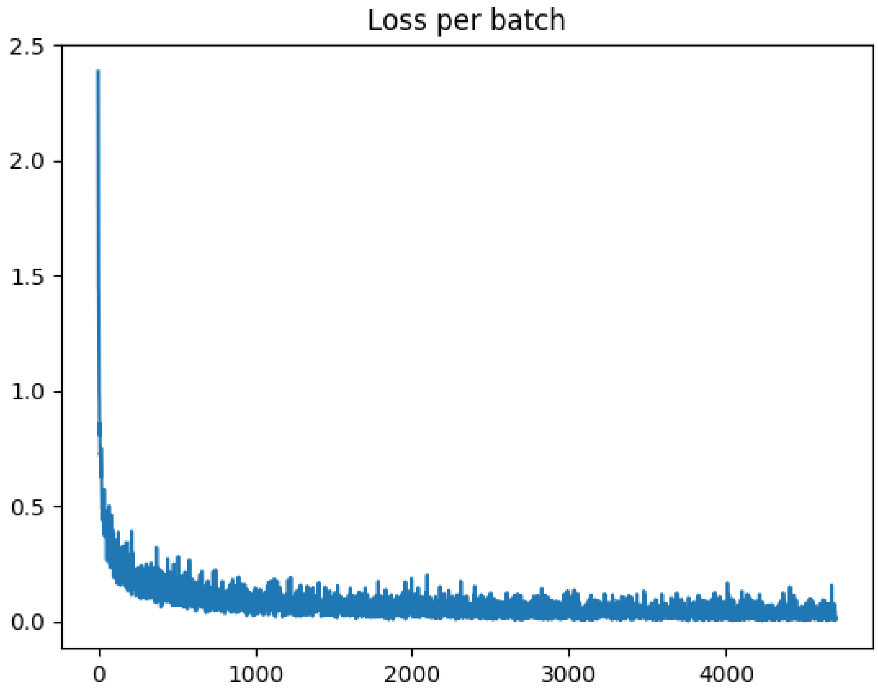
Metrics

- Loss
- Accuracy

Plot Batch Loss

```
def plot_losses(losses):
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot(losses)
    ax.set_title("Loss per batch")
    fig.show()
    plt.savefig('output.png')
```

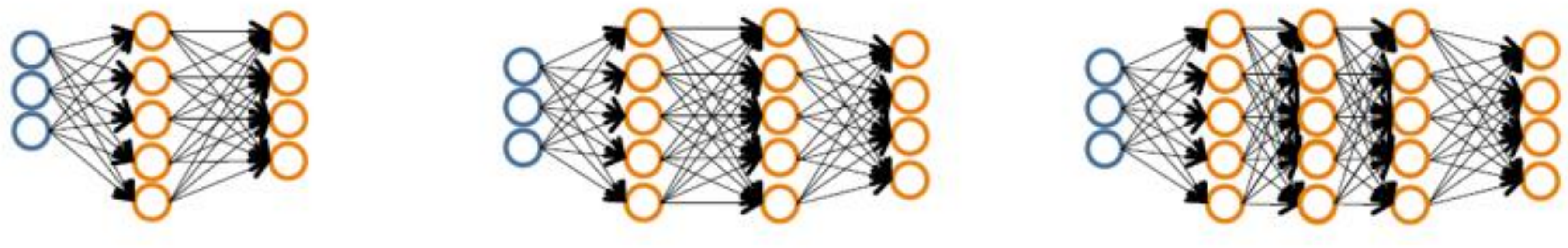
```
plot_losses(losses)
```



Outline

- Training with backpropagation
  - Gradient-Descent Algorithm
  - Derivation of gradients
  - Stochastic and Mini-Batch Gradient Descent
- Lab
  - MNIST dataset
- Regularization for Neural Networks
  - L2/L1 regularization
  - Dropout

Overfitting



- The larger the network, the higher the capacity (more model parameters)
- But also more prone to overfitting!

Regularization

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) \quad + \lambda R(W)$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

λ = regularization strength (hyperparameter)

L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$ → Weight decay

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

- When computing gradients of loss function, regularization term needs to be taken into account

Dropout

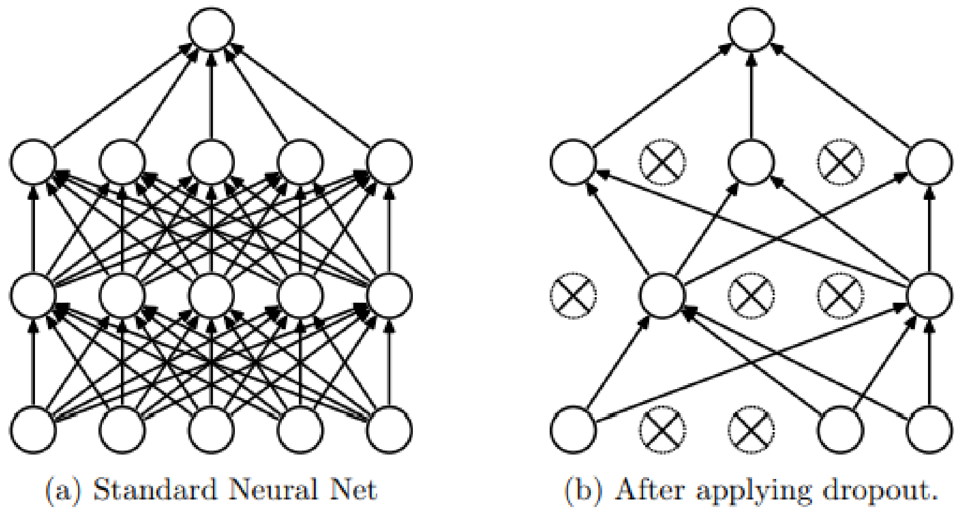


Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

- Regularization technique that has proven very effective for deep learning
- Srivastava et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research 15, 2014

Dropout

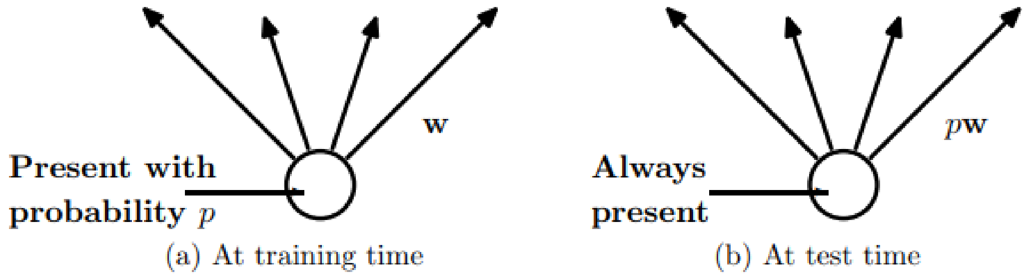
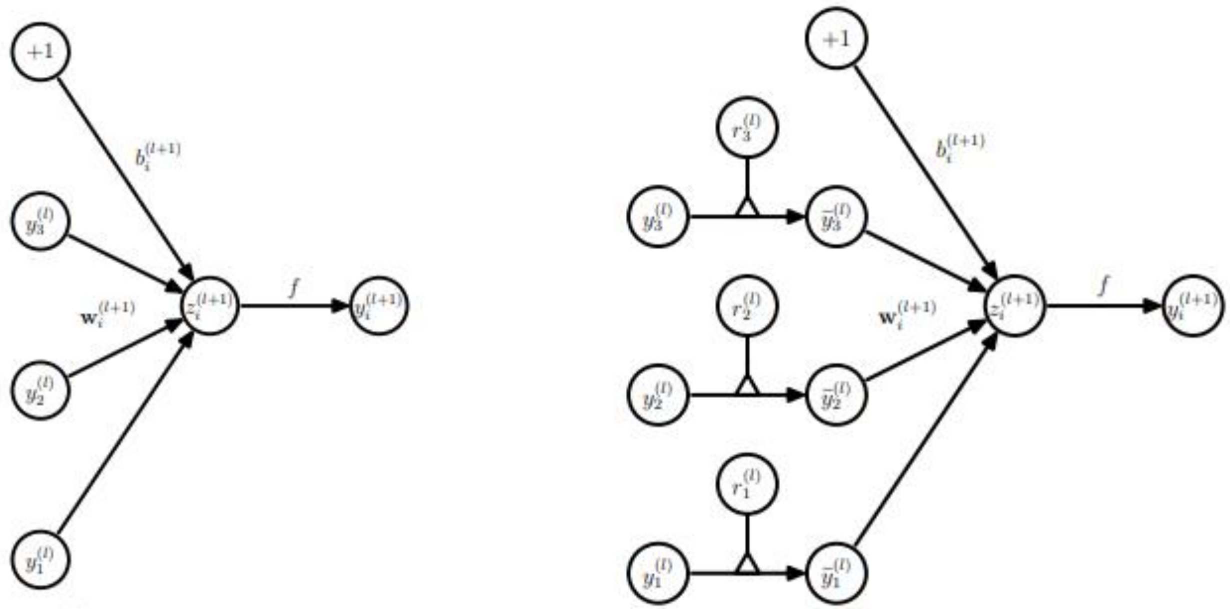


Figure 2: **Left**: A unit at training time that is present with probability p and is connected to units in the next layer with weights $\mathbf{w}$. **Right**: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

- At training time, sample a sub-network and learn weights
  - Keep each neuron with probability p
- At testing time, all neurons are there, but reduce weight by a factor of p

Dropout



(a) Standard network

(b) Dropout network

Figure 3: Comparison of the basic operations of a standard and dropout network.

- $r_i^{(\ell)}$ Is a random variable taking value 1 with probability p and value 0 with probability 1-p
- Multiply output of each layer by $r_i^{(\ell)}$

Results on MNIST

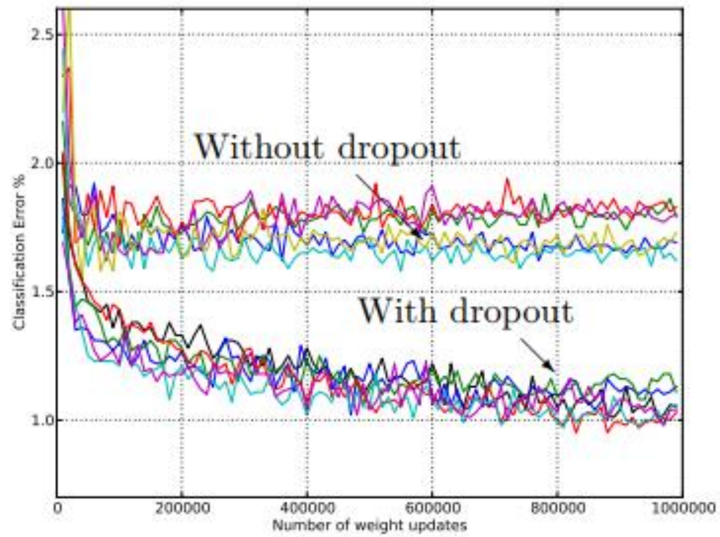


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

Hyper-parameter learning

- Architecture
  - Number layers, hidden units, activation functions
- Regularization
- Learning rate

- Can tune hyper-parameters with cross-validation
- More advanced techniques: meta learning

Acknowledgements

- Slides made using resources from:
  - Yann LeCun
  - Andrew Ng
  - Eric Eaton
  - David Sontag
  - Andrew Moore
- Thanks!