W4995 Applied Machine Learning
Fall 2021

Lecture 8

Dr. Vijay Pappu

Announcements

- Project deliverable 2 due on 11/10
- HW2 grades will be released this week
- Midterm grades will be released next week
- HW3 will be posted this week

In today's lecture, we will cover...

- Neural Networks
- Advanced Neural Networks

Neural Networks

Neural Networks

- Neural networks existed since 1960s
- Several successful applications existed in 1990s using Neural Networks
    - Automatically sorting mail at USPS by identifying zip codes
    - Automatic reading of checks from ATMs
- In last decade, some innovations accelerated Neural Networks research
    - Storage & access to larger datasets
    - Faster computation using GPUs

Neural Networks

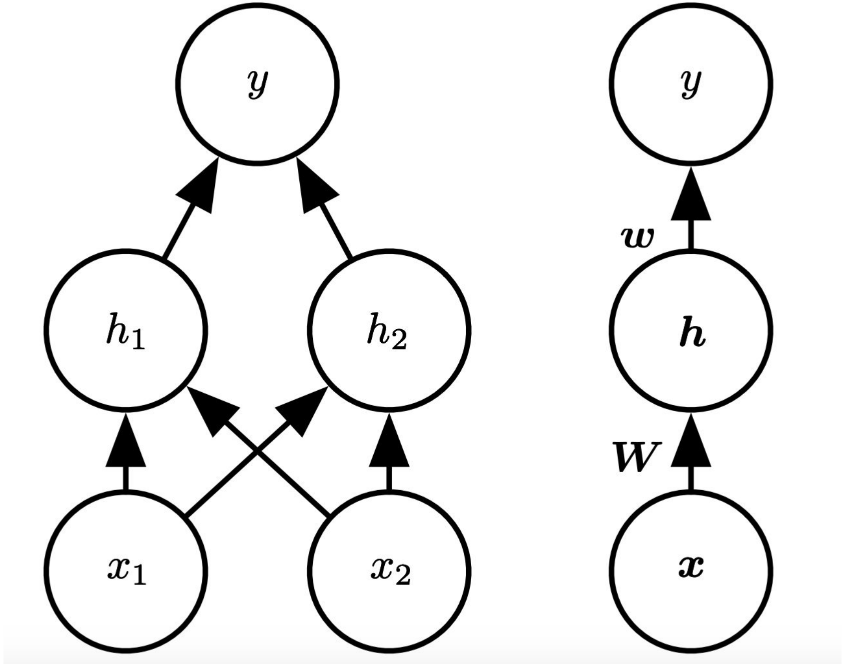
- Several improvements over the last decade
  - Dropout
  - Batch normalization
  - Non-linear activation functions
  - Optimization algorithms (Adam, ..., etc.)
  - Residual networks

Basic Neural Network Architecture

Outputs

Hidden layer

Inputs



Activation function

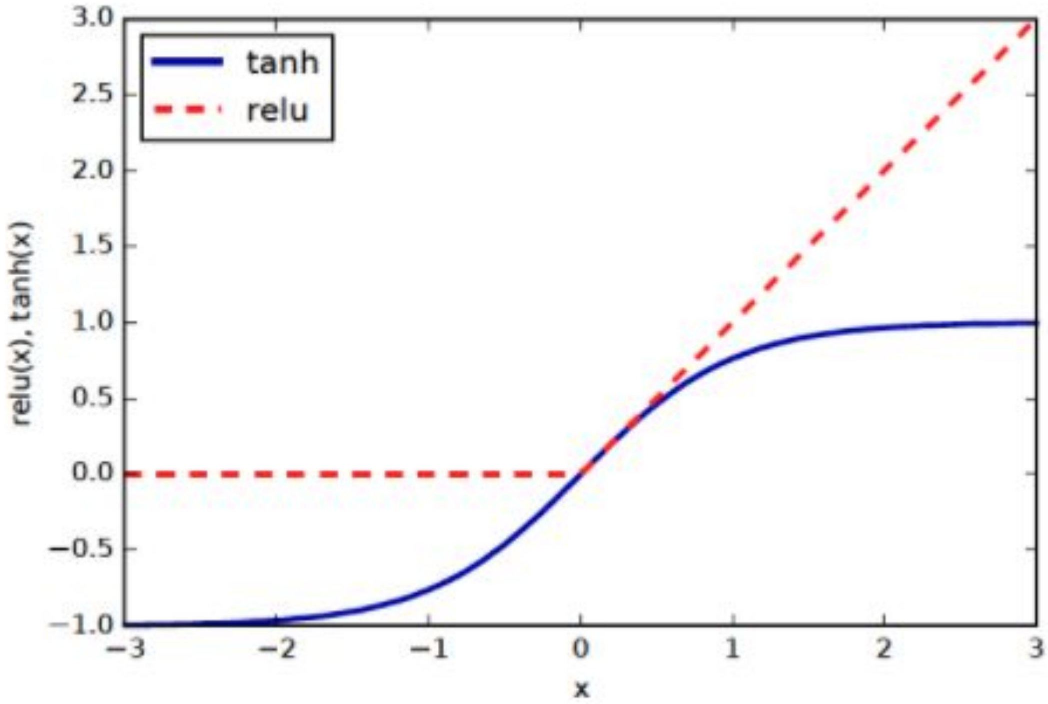
Weight matrix

$$h(x) = f\left(Wx + b_1\right) \qquad o(x) = g\left(wh(x) + b_2\right)$$

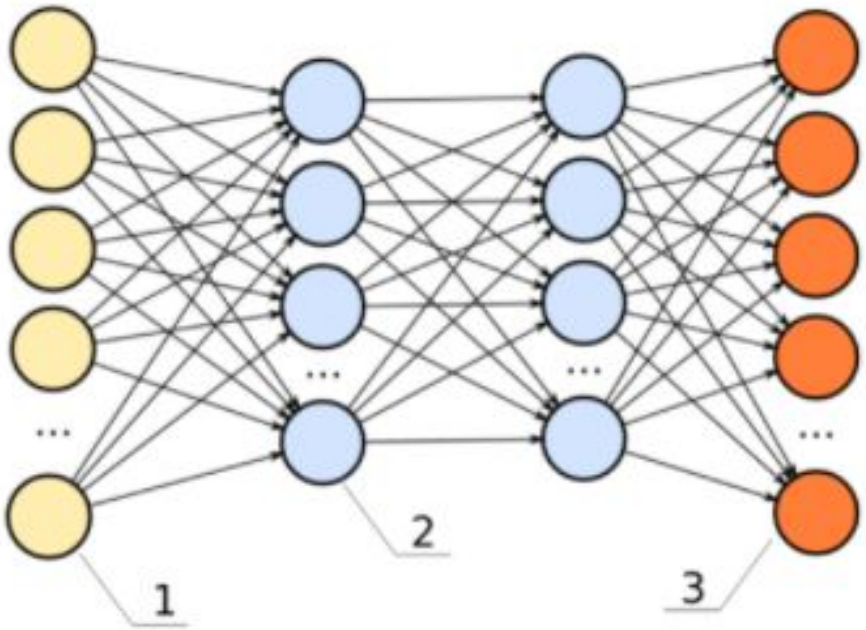
Neural Networks - Activation function

- A linear activation function would reduce the neural network to a linear model
- Non-linearity is introduced into neural networks using a non-linear activation function
- Common activation functions include tanh, Rectified Linear Unit (ReLU) and sigmoid.
- ReLU is the most popular of the activation functions

Neural Networks - Activation function



Deep Neural Network Architecture

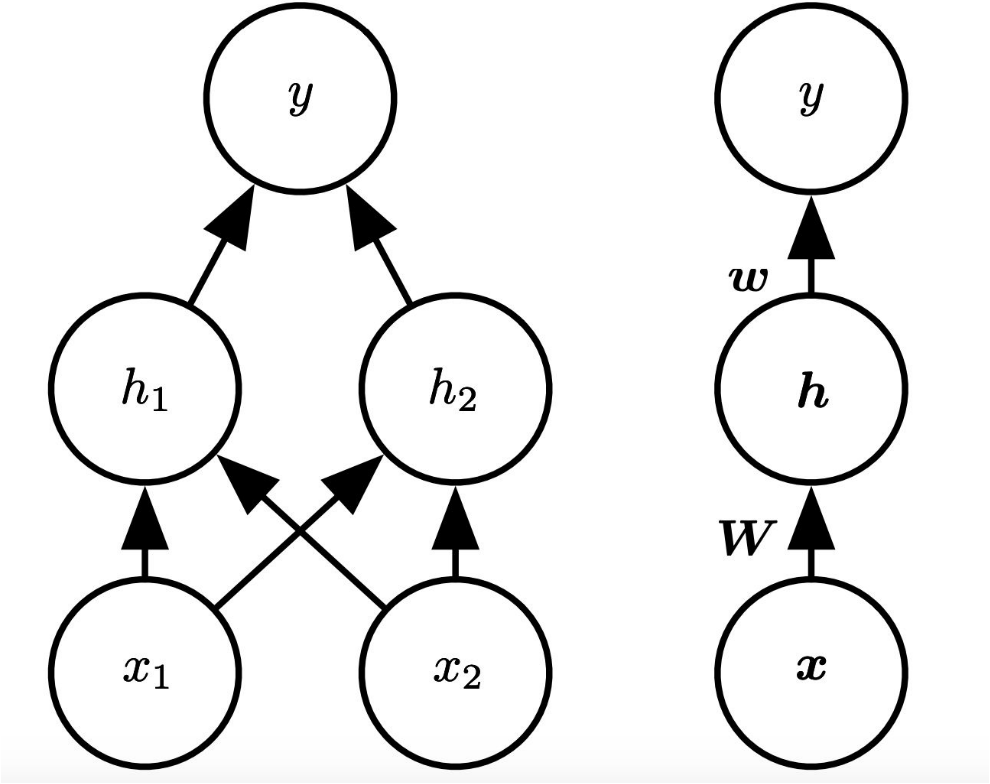


Linear Regression as Neural Network

Outputs

Hidden layer

Inputs

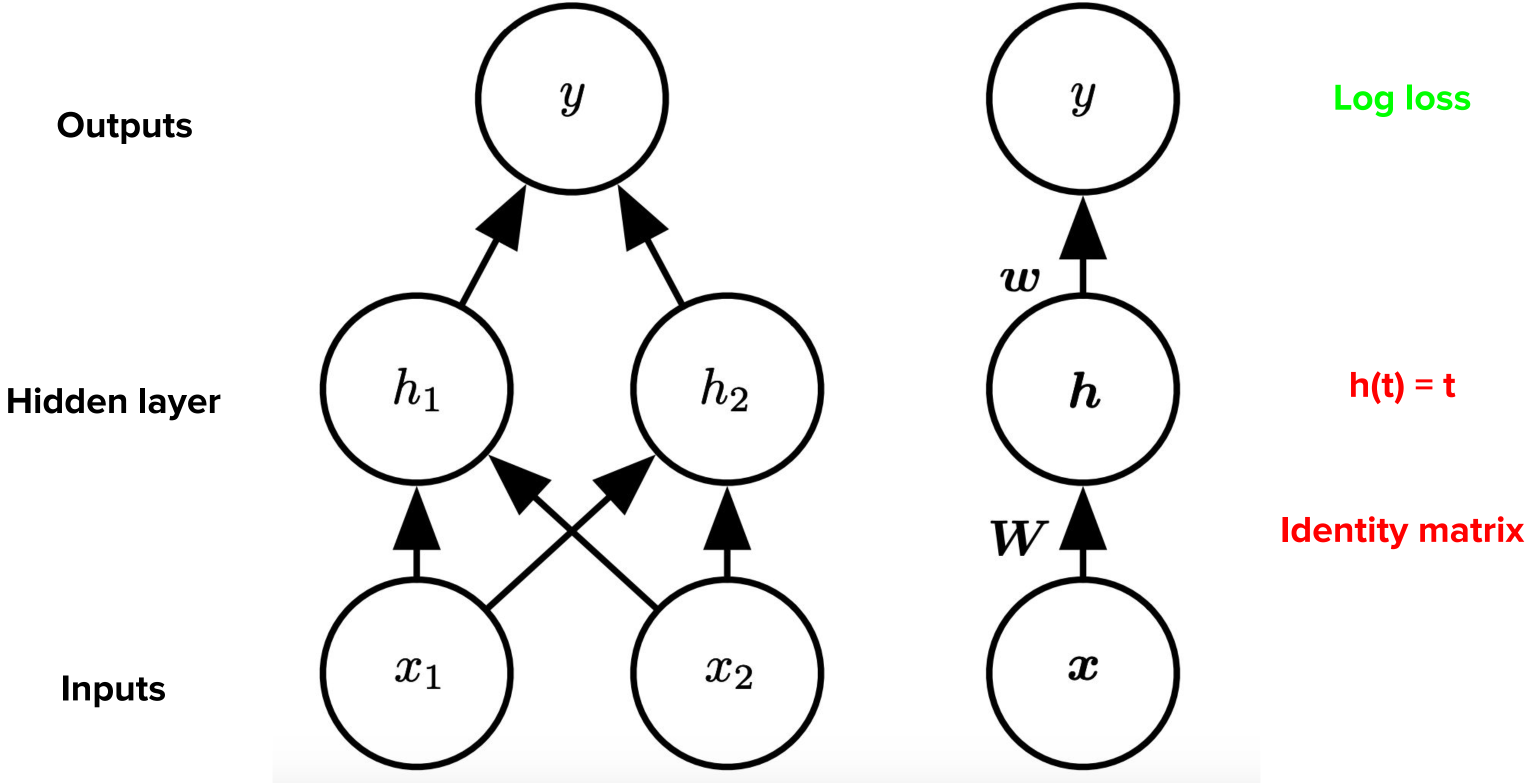


Squared loss

h(t) = t

Identity matrix

Linear Regression as Neural Network

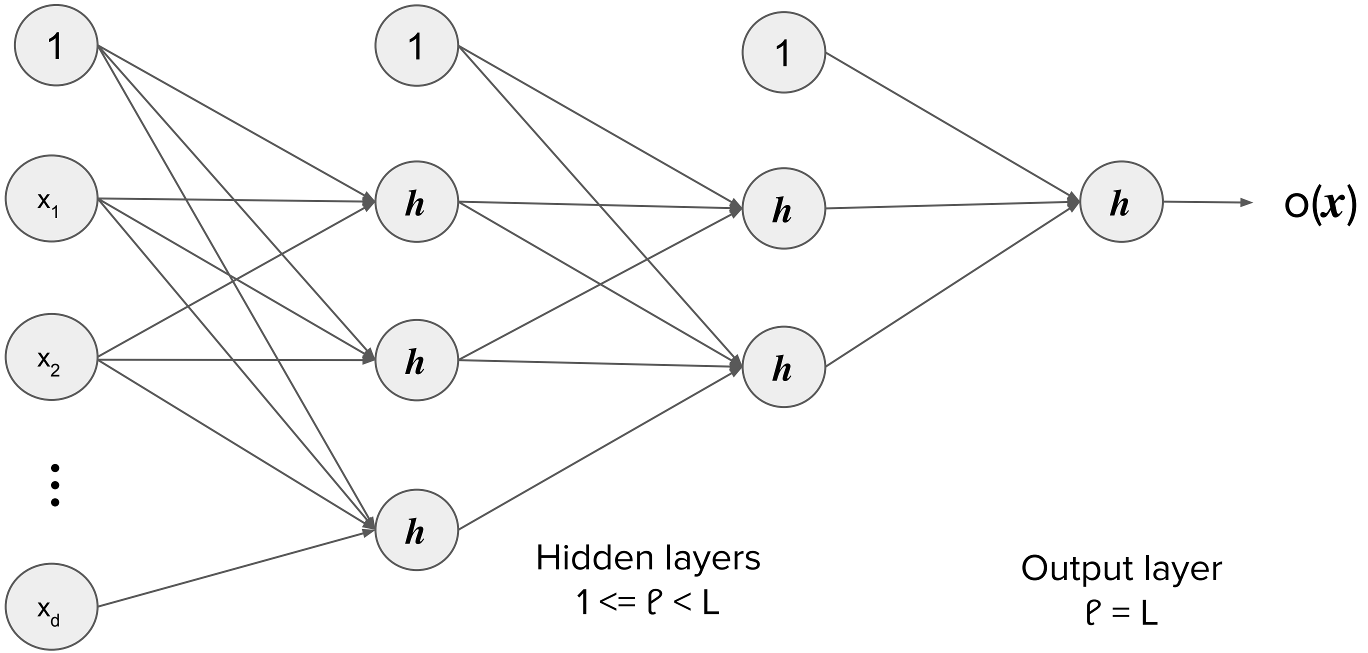


Neural Networks

- Generally applicable to both regression and classification problems
- Universal approximators and hence overfit
- Non-convex optimization (with non-linear activation function)
- Works well with large datasets
- Very slow to train on CPUs (better on GPUs)
- Special frameworks (PyTorch, Tensorflow etc.) to train
- Requires pre-processing since training involves dot product computations

Training Neural Networks

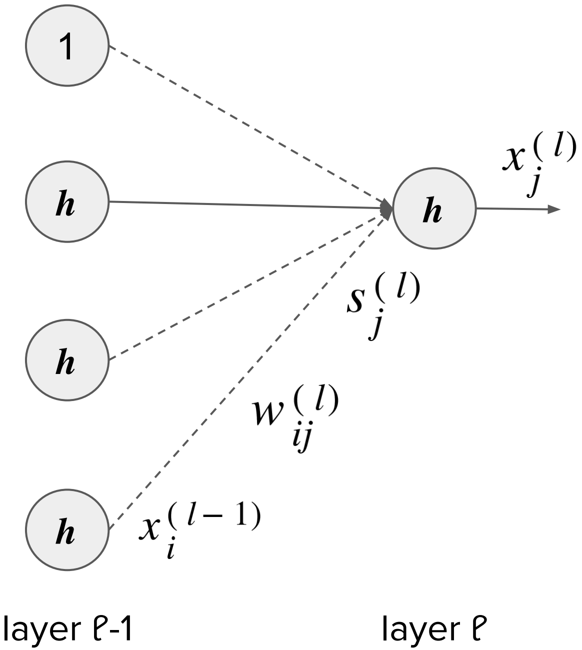
Neural Networks



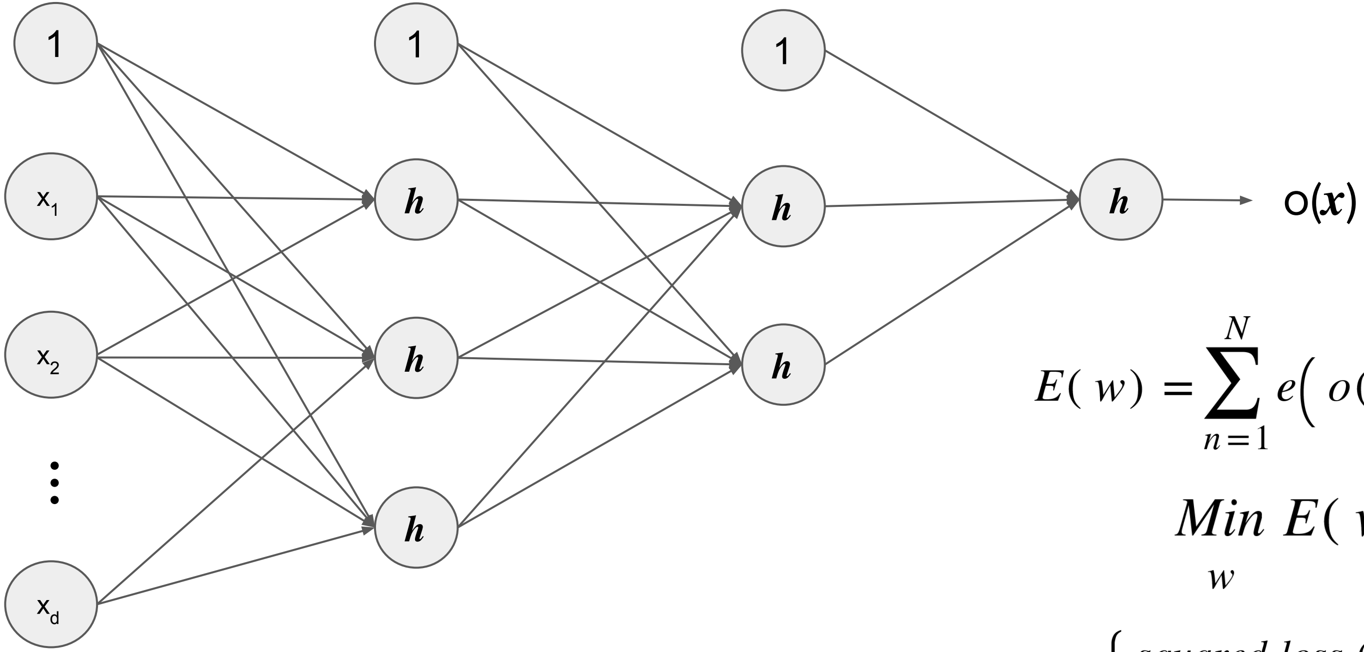
Neural Networks - Notation

$$w_{ij}^{(l)} \rightarrow \begin{cases} 1 \leq l \leq L & \text{layers} \\ 0 \leq i \leq d^{(l-1)} & \text{inputs} \\ 1 \leq j \leq d^{(l)} & \text{outputs} \end{cases}$$

$$x_j^{(l)} = h(s_j^{(l)}) = h\left(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}\right)$$



Neural Networks - Objective



$$E(w) = \sum_{n=1}^{N} e\Big(o(\boldsymbol{x}_{\mathrm{n}}),\ \boldsymbol{y}_{\mathrm{n}}\Big)$$

$$\underset{w}{Min}\ E(w)$$

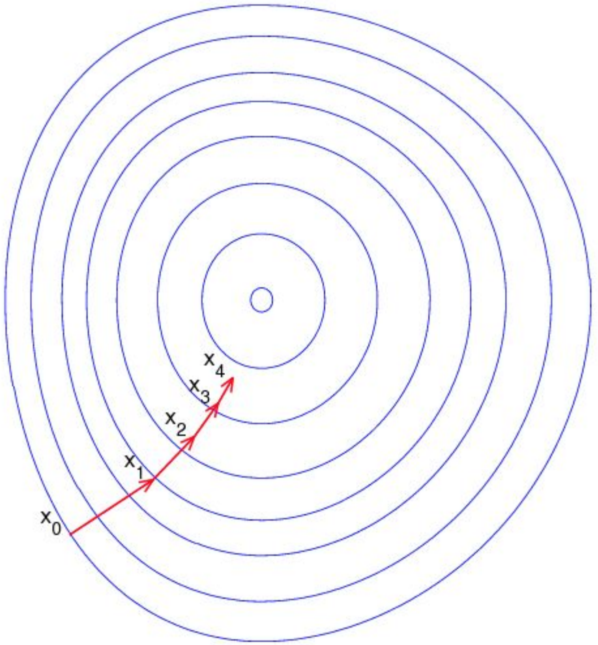
$$e \rightarrow \begin{cases} squared\ loss\ (regression) \\ log\ loss\ (classification) \end{cases}$$

Neural Networks - Gradient Descent (GD)

$$\underset{w}{Min}\ E(w)$$

$$w^{(i+1)} = w^{i} - \eta \nabla E(w^{i})$$

learning rate



converges to a local minimum

https://builtin.com/data-science/gradient-descent

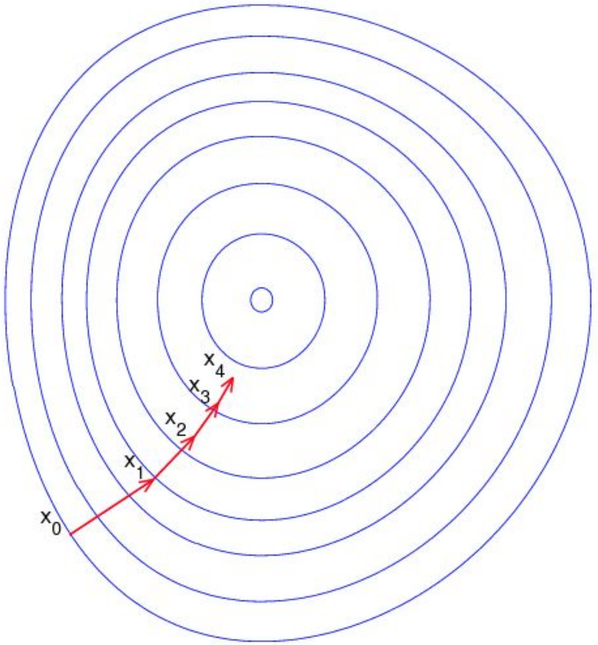
Neural Networks - Gradient Descent (GD)

$$\underset{w}{Min}\ E(w)$$

$$w^{(i+1)} = w^{i} - \eta \boxed{\nabla E(w^{i})}$$

learning rate

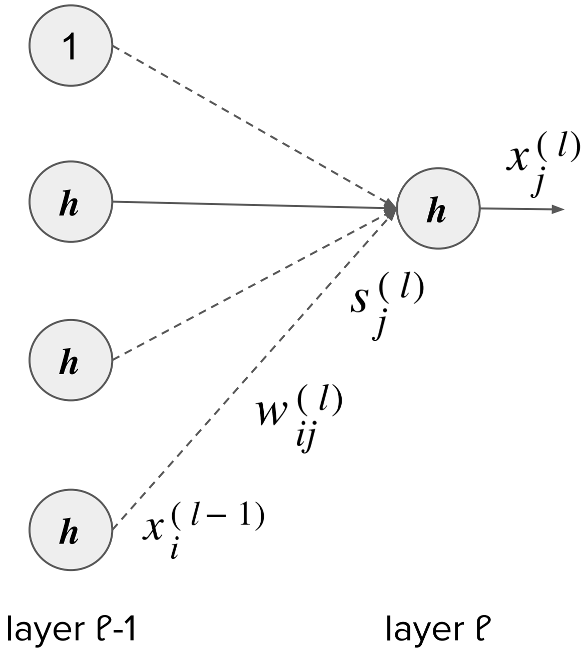
converges to a local minimum



https://builtin.com/data-science/gradient-descent

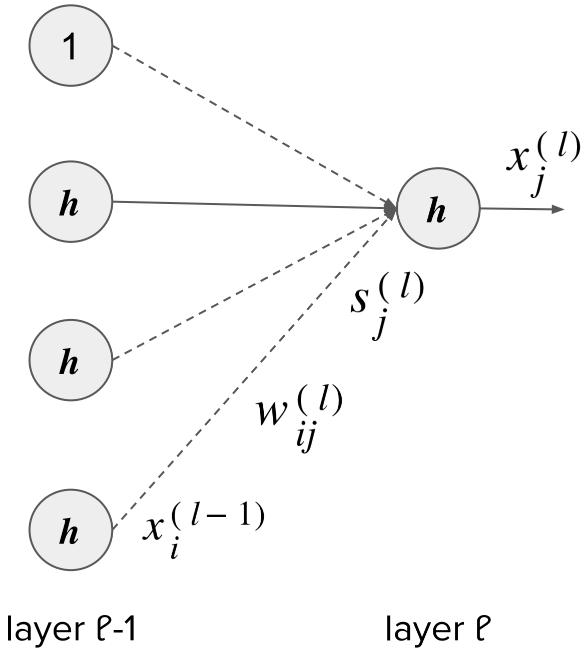
Neural Networks - Computing Gradient

$$\frac{\partial E(w)}{\partial w_{ij}^{(l)}}$$



Neural Networks - Computing Gradient

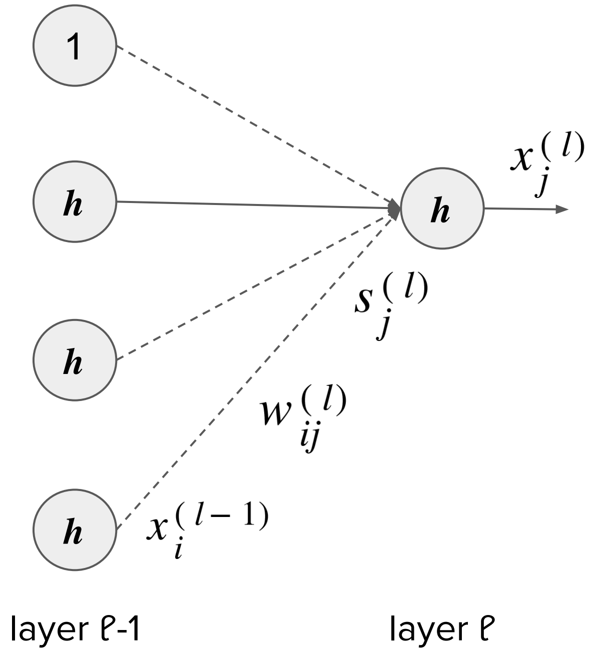
$$\frac{\partial E(w)}{\partial w_{ij}^{(l)}} = \frac{\partial E(w)}{\partial s_j^{(l)}} \times \frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}}$$



Neural Networks - Computing Gradient

$$\frac{\partial E(w)}{\partial w_{ij}^{(l)}} = \frac{\partial E(w)}{\partial s_j^{(l)}} \times \frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}}$$

$$s_j^{(l)} = \sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}$$

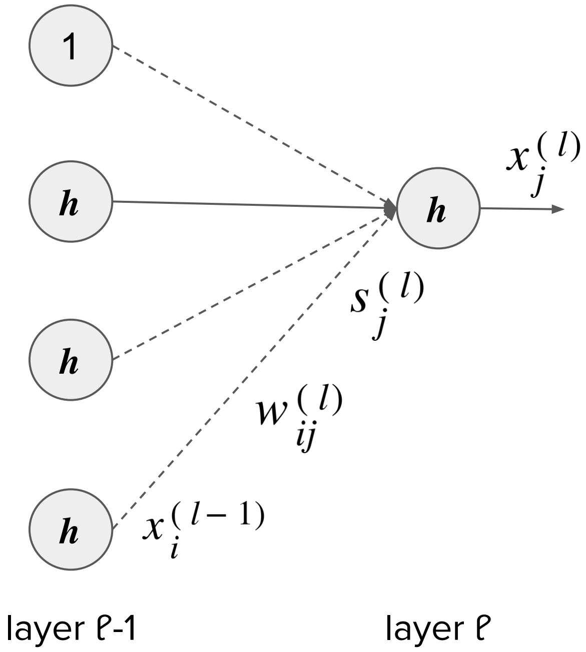


Neural Networks - Computing Gradient

$$\frac{\partial E(w)}{\partial w_{ij}^{(l)}} = \frac{\partial E(w)}{\partial s_j^{(l)}} \times \frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}}$$

$$s_j^{(l)} = \sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}$$

$$\frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}} = x_i^{(l-1)}$$

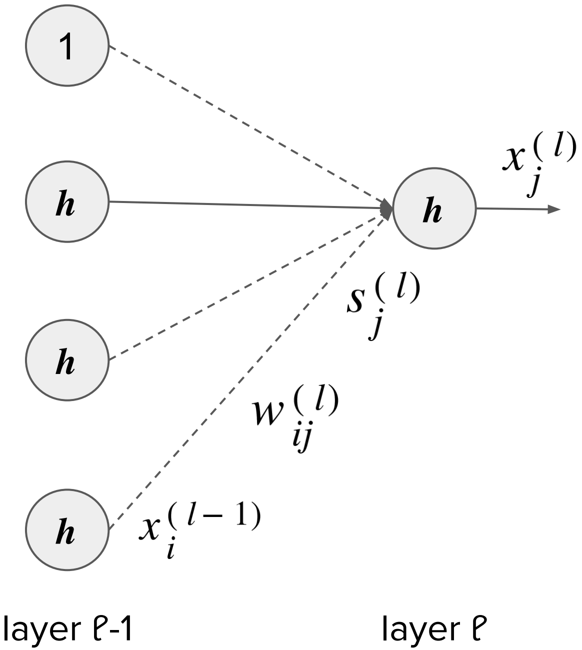


Neural Networks - Computing Gradient

$$\frac{\partial E(w)}{\partial w_{ij}^{(l)}} = \frac{\partial E(w)}{\partial s_j^{(l)}} \times \frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}}$$

$$\delta_i^{(l)} \qquad s_j^{(l)} = \sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}$$

$$\frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}} = x_i^{(l-1)}$$

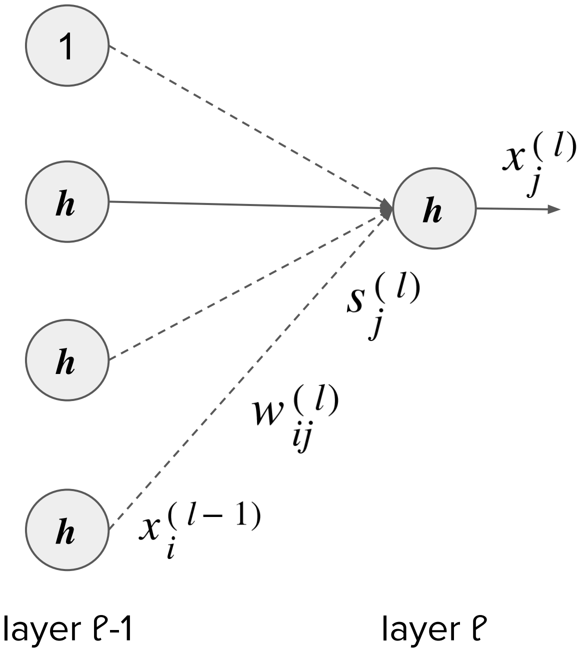


Neural Networks - Computing Gradient

$$\frac{\partial E(w)}{\partial w_{ij}^{(l)}} = \frac{\partial E(w)}{\partial s_j^{(l)}} \times \frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}}$$

$$\color{red}{\delta_i^{(l)}} \qquad s_j^{(l)} = \sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}$$

? $$\frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}} = x_i^{(l-1)}$$

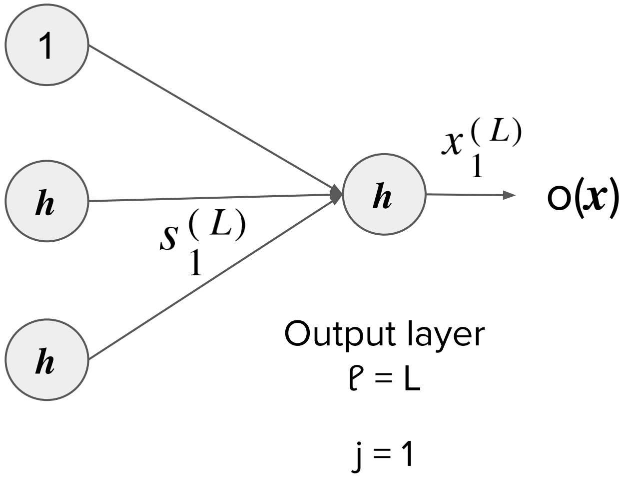


Neural Networks - Computing Gradient

$$\delta_j^{(l)} = \frac{\partial E(w)}{\partial s_j^{(l)}}$$

Neural Networks - Computing Gradient

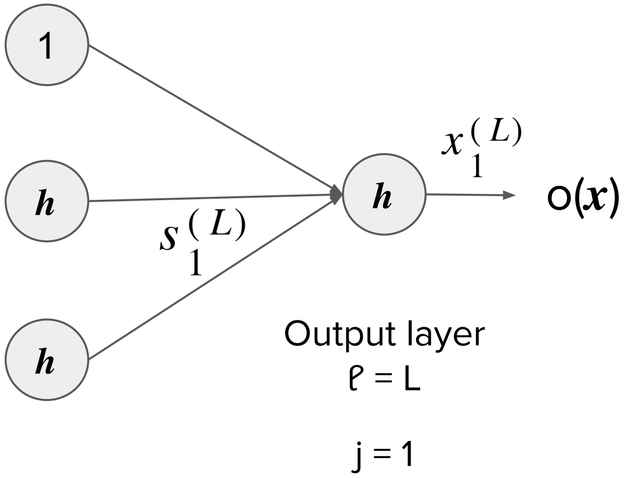
$$\delta_j^{(l)} = \frac{\partial E(w)}{\partial s_j^{(l)}}$$



Neural Networks - Computing Gradient

$$\delta_j^{(l)} = \frac{\partial E(w)}{\partial s_j^{(l)}}$$

$$\delta_1^{(L)} = \frac{\partial E(w)}{\partial s_1^{(L)}} \quad \text{(final layer)}$$

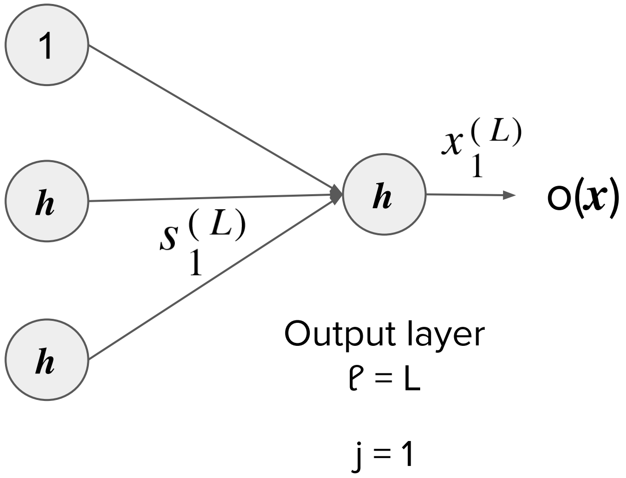


Neural Networks - Computing Gradient

$$\delta_j^{(l)} = \frac{\partial E(w)}{\partial s_j^{(l)}}$$

$$\delta_1^{(L)} = \frac{\partial E(w)}{\partial s_1^{(L)}}$$ (final layer)

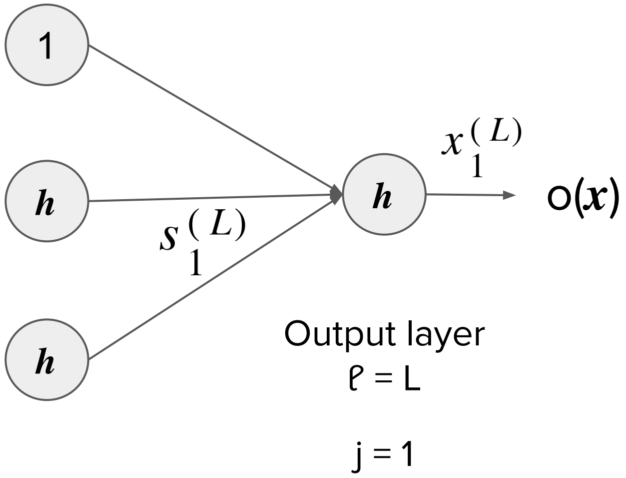
$$E(w) = \sum_{n=1}^{N} e\left(o(x_n),\ y_n\right)$$



Neural Networks - Computing Gradient

$$\delta_j^{(l)} = \frac{\partial E(w)}{\partial s_j^{(l)}}$$

$$\delta_1^{(L)} = \frac{\partial E(w)}{\partial s_1^{(L)}} \quad \text{(final layer)}$$



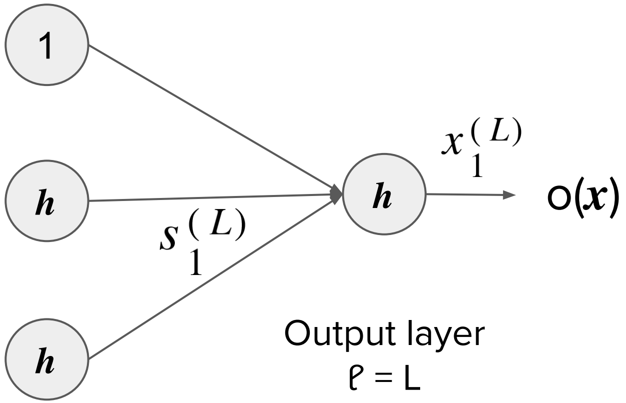
$$E(w) = \sum_{n=1}^{N} e\left(o(x_n),\ y_n\right)$$

$$e(w) = \left(x_1^{(\mathrm{L})} - y_n\right)^2$$

Neural Networks - Computing Gradient

$$\delta_j^{(l)} = \frac{\partial E(w)}{\partial s_j^{(l)}}$$

$$\delta_1^{(L)} = \frac{\partial E(w)}{\partial s_1^{(L)}}$$ (final layer)



Output layer
$\ell = L$

j = 1

$$E(w) = \sum_{n=1}^{N} e\big(o(\boldsymbol{x}_\mathrm{n}),\, y_\mathrm{n}\big)$$

$$\boldsymbol{x}_1^{(\mathrm{L})} = h\big(s_1^{(L)}\big)$$

$$e(w) = \big(\boldsymbol{x}_1^{(\mathrm{L})} - y_\mathrm{n}\big)^2$$

Neural Networks - Computing Gradient

$$\delta_i^{(l-1)} = \frac{\partial E(w)}{\partial s_i^{(l-1)}}$$

Neural Networks - Computing Gradient

$$\delta_i^{(l-1)} = \frac{\partial E(w)}{\partial s_i^{(l-1)}}$$

$$= \sum_{j=1}^{d^{(l)}} \frac{\partial E(w)}{\partial s_j^{(l)}} \times \frac{\partial s_j^{(l)}}{\partial x_i^{(l-1)}} \times \frac{\partial x_i^{(l-1)}}{\partial s_i^{(l-1)}}$$

Neural Networks - Computing Gradient

$$
\begin{aligned}
\delta_i^{(l-1)} &= \frac{\partial E(w)}{\partial s_i^{(l-1)}} \\
&= \sum_{j=1}^{d^{(l)}} \frac{\partial E(w)}{\partial s_j^{(l)}} \times \frac{\partial s_j^{(l)}}{\partial x_i^{(l-1)}} \times \frac{\partial x_i^{(l-1)}}{\partial s_i^{(l-1)}} \\
&= \sum_{j=1}^{d^{(l)}} \delta_j^{(l)} \times w_{ij}^{(l)} \times h'\left(s_i^{(l-1)} \right)
\end{aligned}
$$

Neural Networks - Computing Gradient

$$\delta_i^{(l-1)} = \frac{\partial E(w)}{\partial s_i^{(l-1)}}$$

$$= \sum_{j=1}^{d^{(l)}} \frac{\partial E(w)}{\partial s_j^{(l)}} \times \frac{\partial s_j^{(l)}}{\partial x_i^{(l-1)}} \times \frac{\partial x_i^{(l-1)}}{\partial s_i^{(l-1)}}$$

$$= \sum_{j=1}^{d^{(l)}} \delta_j^{(l)} \times w_{ij}^{(l)} \times h'\left(s_i^{(l-1)}\right)$$

$$\delta_j^{(l-1)} = h'\left(s_i^{(l-1)}\right) \sum_{j=1}^{d^{(l)}} w_{ij}^{(l)} \delta_j^{(l)}$$

Neural Networks - Back Propagation Algorithm

Initialize all weights $w_{ij}^{(l)}$ at random
for $t = 0, 1, 2, \ldots$ **do**
    Pick $n \in \{1, 2, \cdots, N\}$
    Forward: Compute all $x_j^{(l)}$
    Backward: Compute all $\delta_j^{(l)}$
    Update the weights: $w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta\, x_i^{(l-1)} \delta_j^{(l)}$
    Iterate to the next step until it is time to stop
Return the final weights $w_{ij}^{(l)}$

https://home.work.caltech.edu/slides/slides10.pdf

Neural Networks Implementation - Batch sizes

$$E(w) = \sum_{n=1}^{N} e\left(o(\boldsymbol{x}_{\mathbf{n}}) ,\ \boldsymbol{y}_{\mathbf{n}} \right)$$

batch $\longrightarrow$ $$w^{(i+1)} = w^{i} - \eta \sum_{n=1}^{N} \frac{\partial e\left(o\left(x_{n} \right), y_{n} \right)}{\partial w^{i}}$$

stochastic $\longrightarrow$ $$w^{(i+1)} = w^{i} - \eta \frac{\partial e\left(o\left(x_{n} \right), y_{n} \right)}{\partial w^{i}}$$

mini-batch $\longrightarrow$ $$w^{(i+1)} = w^{i} - \eta \sum_{j=k}^{k+m} \frac{\partial e\left(o\left(x_{j} \right), y_{j} \right)}{\partial w^{i}}$$

Neural Networks Implementation - Learning Rate

- Learning rate is one of the important parameters during neural network training
- A constant learning rate is not necessarily a good choice
- Learning rate adaptive to # of iterations is a good choice
- Learning rates per parameter is currently state-of-the-art

Neural Networks Implementation - Optimization Algorithms

- Several optimization algorithms with adaptive learning rates have been proposed recently for training neural networks
- Popular algorithms include Adam, RmsProp, Adagrad and momentum
- Adaptive learning rate methods work well for sparse data
- These methods don't necessarily require any tuning (and work out-of-the-box)
- Overall, [Adam] seems to be best choice for training neural networks

https://ruder.io/optimizing-gradient-descent/

Neural Networks - Hyperparameters

- Network parameters
  - # hidden layers
  - # nodes per layer
- Activation function (ReLU, tanh, sigmoid etc.)
- Weight initialization
- Batch size
- # of epochs
- Optimization algorithm parameters

Neural Networks - Example

```
data = pd.read_csv('ml-100k/u.data', sep="\t", header=None)
data.columns = ["user_id", "item_id", "rating", "timestamp"]
data.drop(["timestamp"], axis=1, inplace=True)
data_processed = pd.get_dummies(data, columns=['user_id', 'item_id'])
data_processed["rating"] = data_processed["rating"] >=4
y = data_processed["rating"]
X = data_processed.drop(["rating"],axis=1)
dev_X, test_X, dev_y, test_y = train_test_split(X, y, test_size=0.2,
                                                random_state=42)
data_processed.head()
```

Neural Networks - Example

	rating	user_id_1	user_id_2	user_id_3	user_id_4	user_id_5	user_id_6	user_id_7	user_id_8	user_id_9	...	item_id_1673	item_id_1674
0	False	0	0	0	0	0	0	0	0	0	...	0	0
1	False	0	0	0	0	0	0	0	0	0	...	0	0
2	False	0	0	0	0	0	0	0	0	0	...	0	0
3	False	0	0	0	0	0	0	0	0	0	...	0	0
4	False	0	0	0	0	0	0	0	0	0	...	0	0

Neural Networks - Example

sklearn.neural_network.MLPClassifier

class sklearn.neural_network.MLPClassifier(*hidden_layer_sizes=(100), activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000*) [source]

```
start_time = time.time()
mlp = MLPClassifier(solver='lbfgs', random_state=0).fit(dev_X, dev_y)
print(f"Training time: {(time.time() - start_time) / 60} mins")
print(mlp.score(dev_X, dev_y))
print(mlp.score(test_X, test_y))
```

```
Training time: 3.9310388525327045 mins
0.8389875
0.6748
```

1 hidden layer with 100 nodes

Neural Networks - Example

```
start_time = time.time()
mlp = MLPClassifier(solver='lbfgs', random_state=0,
                    hidden_layer_sizes=(100, 100, 100))
mlp.fit(dev_X, dev_y)
print(f"Training time: {(time.time() - start_time) / 60} mins")
print(mlp.score(dev_X, dev_y))
print(mlp.score(test_X, test_y))
```

```
Training time: 5.745137381553 65 mins
0.73425
0.70735
```

3 hidden layers with 100 nodes

Neural Networks - Example

```
start_time = time.time()
mlp = MLPClassifier(solver='lbfgs', random_state=0,
                    verbose=True, activation='tanh')
mlp.fit(dev_X, dev_y)
print(f"Training time: {(time.time() - start_time) / 60} mins")
print(mlp.score(dev_X, dev_y))
print(mlp.score(test_X, test_y))
```

```
Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
*****    200    242      1     0     0   3.653D-03   5.043D-01
  F =  0.50434270618298360

STOP: TOTAL NO. of ITERATIONS REACHED LIMIT
Training time: 4.443095231056214 mins
0.7505
0.7055
```

Neural Networks - Example

```
start_time = time.time()
mlp = MLPClassifier(solver='lbfgs', random_state=0,
                    activation='tanh', hidden_layer_sizes=(100, 100, 100))
mlp.fit(dev_X, dev_y)
print(f"Training time: {(time.time() - start_time) / 60} mins")
print(mlp.score(dev_X, dev_y))
print(mlp.score(test_X, test_y))
```

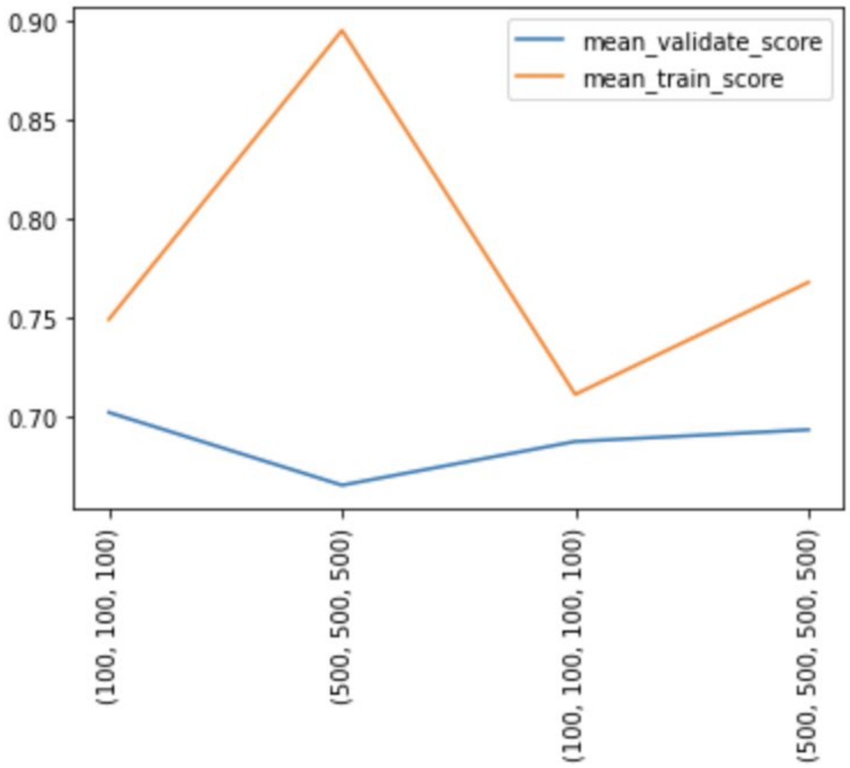
```
Training time: 6.259653635819753 mins
0.7574125
0.71345
```

Neural Networks - Hyperparameter tuning

```
start_time = time.time()
pipe = make_pipeline(MLPClassifier(solver="lbfgs",random_state=0))
param_grid = {'mlpclassifier__hidden_layer_sizes':
              [(100, 100, 100), (500, 500, 500),
               (100, 100, 100, 100), (500, 500, 500, 500),
              ]
             }
grid = GridSearchCV(pipe, param_grid, cv=3,
                    return_train_score=True)
grid.fit(dev_X, dev_y)
print(f"Training time: {(time.time() - start_time) / 60} mins")
```

```
Training time: 144.73998938798906 mins
```

Neural Networks - Hyperparameter tuning



Questions?

Let’s take a 10 min break!

Advanced Neural Networks

Neural Networks Training

- Training neural networks could be painstakingly slow (shown in example before)
  - Gradient estimations
  - Learned parameters
- Some improvements have surprisingly improved the training times:
  - Autodiff for gradient estimation
  - Leveraging GPUs for matrix computations

Neural Networks Training - Autodiff

- Autodiff efficiently estimates the gradients while training the neural networks
- It relies on the principle that most functions can be composed of elementary functions like *e^x*, *cos(x)*, *or* x^2.
- Autodiff leverage chain-rule w.r.t elementary functions to derive the derivative of the function
- Autodiff creates a **computation graph** that allows it to efficiently estimate the gradient of any function.

Neural Networks Training - Autodiff

$$f(x, y, z) = xy + z$$

https://towardsdatascience.com/build-your-own-automatic-differentiation-program-6ecd585eec2a

Neural Networks Training - Autodiff

$$f(x, y, z) = xy + z$$

$$multiply(x, y) = xy$$

$$add(x, y) = x + y$$

https://towardsdatascience.com/build-your-own-automatic-differentiation-program-6ecd585eec2a

Neural Networks Training - Autodiff

$$f(x, y, z) = xy + z$$

$$multiply(x, y) = xy$$

$$add(x, y) = x + y$$

$$f(x, y, z) = add(multiply(x, y), z)$$

https://towardsdatascience.com/build-your-own-automatic-differentiation-program-6ecd585eec2a

Neural Networks Training - Autodiff

$$f(x, y, z) = xy + z$$

$$multiply(x, y) = xy$$

$$add(x, y) = x + y$$

$$f(x, y, z) = add(multiply(x, y), z)$$

$$\frac{\partial f(x, y, z)}{\partial x} = \frac{\partial add(multiply(x, y), z)}{\partial multiply(x, y)} \times \frac{\partial multiply(x, y)}{\partial x}$$

https://towardsdatascience.com/build-your-own-automatic-differentiation-program-6ecd585eec2a

Neural Networks Training - Autodiff

$$f(x, y, z) = xy + z$$

$$multiply(x, y) = xy$$

$$add(x, y) = x + y$$

$$f(x, y, z) = add(multiply(x, y), z)$$

$$\frac{\partial f(x, y, z)}{\partial x} = \frac{\partial add(multiply(x, y), z)}{\partial multiply(x, y)} \times \frac{\partial multiply(x, y)}{\partial x}$$

$$1 \quad \times \quad y$$

https://towardsdatascience.com/build-your-own-automatic-differentiation-program-6ecd585eec2a

Neural Networks Training - Autodiff

$$f(x, y, z) = xy + z$$

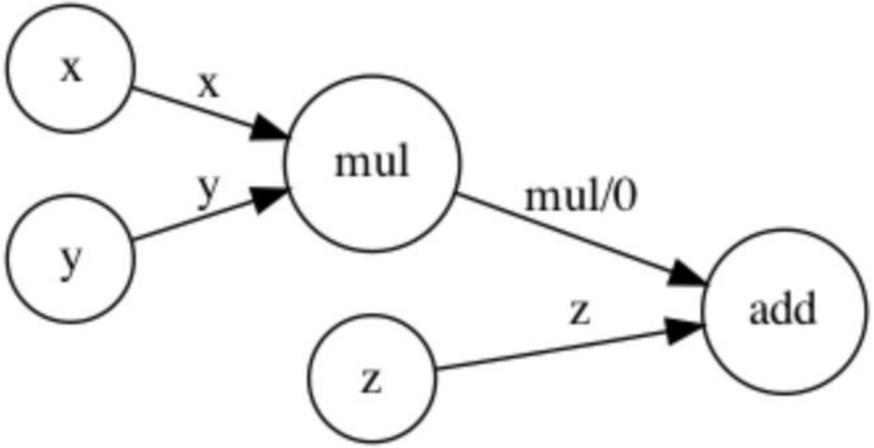
$$multiply(x, y) = xy$$

$$add(x, y) = x + y$$

$$f(x, y, z) = add(multiply(x, y), z)$$

$$\frac{\partial f(x, y, z)}{\partial x} = \frac{\partial add(multiply(x, y), z)}{\partial multiply(x, y)} \times \frac{\partial multiply(x, y)}{\partial x}$$

$$1 \times y$$



https://towardsdatascience.com/build-your-own-automatic-differentiation-program-6ecd585eec2a

Neural Networks Training - GPUs

- Neural networks training often involves matrix operations on big matrices
- CPUs aren’t particularly suited for such operations
- Graphical Processing Units (GPUs) have been popular in the gaming industry.
- Recently, they have been used to accelerate Neural networks training
- Three things that GPUs excel as compared to CPUs are:
    - Parallelization
    - Larger memory bandwidth
    - Faster memory access

Deep Learning Frameworks

Deep Learning Frameworks

- Autodiff
- GPU support
- Computation graph optimization & inspection
- Data-parallel training on multiple GPUs and/or cluster

Deep Learning Frameworks

- Theano (Tensorflow)
- Torch
- Chainer
- MXNet

Deep Learning Libraries

- Keras (tensorflow, CNTK, theano)
- PyTorch (torch)
- Chainer (chainer)
- MXNet (MXNet)

Deep Learning Libraries - Tensorflow

- Tensorflow is a low-level DL library developed by Google
- TF allows more flexibility and has been particular used in production
- Not particularly useful for research
- Three steps generally involved for training DL models:
  - Build computation graph
  - Create optimizer for computation graph
  - Run computation
- Eager mode (default in TF 2.0)
  - Allows writing imperative code directly

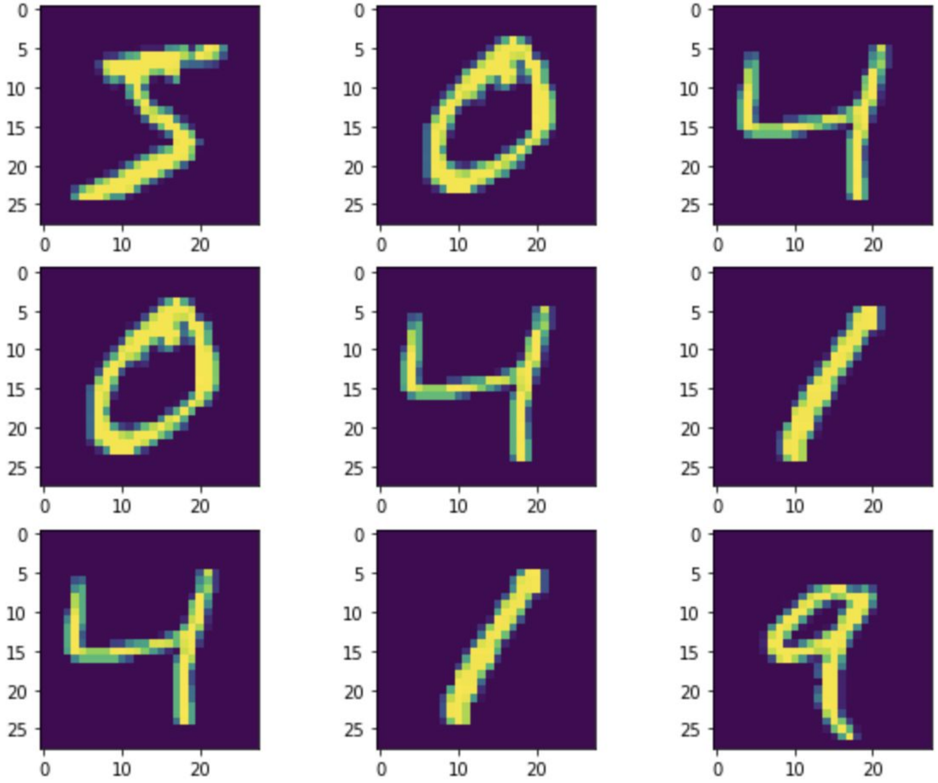
Deep Learning Libraries - PyTorch

- PyTorch is a *high-level* DL library developed by Facebook
- PyTorch is an optimized tensor library for deep learning using GPUs and CPUs.
- Has been particularly useful for research
- Different modules like autograd, optim and nn are defined for flexibility

Deep Learning Libraries - Keras

- Keras is a high-level API that supports multiple backends like TF, Theano, CNTK etc.
- It addition to standard neural networks, it also support convolutional and recurrent neural networks
- Keras allows for distributed training of deep learning models on clusters of GPUs and TPUs

Deep Learning Libraries - Keras example



https://keras.io/api/datasets/mnist/

Deep Learning Libraries - Keras example

```
(X_dev, y_dev), (X_test, y_test) = mnist.load_data()
X_dev = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
X_dev = X_train.astype('float32')
X_test = X_test.astype('float32')
X_dev /= 255
X_test /= 255
num_classes = 10
y_dev = np_utils.to_categorical(y_dev, num_classes)
y_test = np_utils.to_categorical(y_test, num_classes)
print(X_dev.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')
```

```
60000 train samples
10000 test samples
```

Deep Learning Libraries - Keras example

```
model = Sequential([
    Dense(32, input_shape=(784,), activation='relu'),
    Dense(10, activation='softmax')
])
```

Deep Learning Libraries - Keras example

```
model = Sequential([
    Dense(32, input_shape=(784,)),
    Activation('relu'),
    Dense(10),
    Activation('softmax')])
```

Deep Learning Libraries - Keras example

```
model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_12 (Dense)            (None, 32)                25120

 activation_6 (Activation)   (None, 32)                0

 dense_13 (Dense)            (None, 10)                330

 activation_7 (Activation)   (None, 10)                0

=================================================================
Total params: 25,450
Trainable params: 25,450
Non-trainable params: 0
_________________________________________________________________
```

Deep Learning Libraries - Keras example

`compile` method

```
Model.compile(
    optimizer="rmsprop",
    loss=None,
    metrics=None,
    loss_weights=None,
    weighted_metrics=None,
    run_eagerly=None,
    steps_per_execution=None,
    **kwargs
)
```

```
model.compile("adam", "categorical_crossentropy", metrics=["accuracy"])
```

Deep Learning Libraries - Keras example

`fit` method

```
Model.fit(
    x=None,
    y=None,
    batch_size=None,
    epochs=1,
    verbose="auto",
    callbacks=None,
    validation_split=0.0,
    validation_data=None,
    shuffle=True,
    class_weight=None,
    sample_weight=None,
    initial_epoch=0,
    steps_per_epoch=None,
    validation_steps=None,
    validation_batch_size=None,
    validation_freq=1,
    max_queue_size=10,
    workers=1,
    use_multiprocessing=False,
)
```

Trains the model for a fixed number of epochs (iterations on a dataset).

Deep Learning Libraries - Keras example

```
model.fit(X_dev, y_dev, batch_size=128, epochs=10, verbose=1)
```

```
Epoch 1/10
469/469 [==============================] - 1s 1ms/step - loss: 2.1194 - accuracy: 0.5331
Epoch 2/10
469/469 [==============================] - 0s 1ms/step - loss: 1.4685 - accuracy: 0.6806
Epoch 3/10
469/469 [==============================] - 1s 1ms/step - loss: 1.0198 - accuracy: 0.7673
Epoch 4/10
469/469 [==============================] - 1s 1ms/step - loss: 0.7905 - accuracy: 0.8140
Epoch 5/10
469/469 [==============================] - 1s 2ms/step - loss: 0.6571 - accuracy: 0.8402
Epoch 6/10
469/469 [==============================] - 1s 1ms/step - loss: 0.5708 - accuracy: 0.8585
Epoch 7/10
469/469 [==============================] - 0s 1ms/step - loss: 0.5116 - accuracy: 0.8702
Epoch 8/10
469/469 [==============================] - 0s 1ms/step - loss: 0.4687 - accuracy: 0.8786
Epoch 9/10
469/469 [==============================] - 0s 986us/step - loss: 0.4370 - accuracy: 0.8842
Epoch 10/10
469/469 [==============================] - 1s 1ms/step - loss: 0.4127 - accuracy: 0.8888
```

Deep Learning Libraries - Keras example

```
model.fit(X_dev, y_dev, batch_size=128, epochs=10, verbose=1, validation_split=.1)
```

```
Epoch 1/10
422/422 [==============================] - 1s 2ms/step - loss: 0.4013 - accuracy: 0.8901 - val_loss: 0.3246 - val_accuracy: 0.9170
Epoch 2/10
422/422 [==============================] - 0s 1ms/step - loss: 0.3869 - accuracy: 0.8935 - val_loss: 0.3128 - val_accuracy: 0.9173
Epoch 3/10
422/422 [==============================] - 1s 1ms/step - loss: 0.3748 - accuracy: 0.8958 - val_loss: 0.3034 - val_accuracy: 0.9195
Epoch 4/10
422/422 [==============================] - 1s 1ms/step - loss: 0.3645 - accuracy: 0.8985 - val_loss: 0.2954 - val_accuracy: 0.9213
Epoch 5/10
422/422 [==============================] - 1s 1ms/step - loss: 0.3556 - accuracy: 0.9000 - val_loss: 0.2875 - val_accuracy: 0.9225
Epoch 6/10
422/422 [==============================] - 1s 1ms/step - loss: 0.3478 - accuracy: 0.9020 - val_loss: 0.2829 - val_accuracy: 0.9222
Epoch 7/10
422/422 [==============================] - 1s 1ms/step - loss: 0.3411 - accuracy: 0.9039 - val_loss: 0.2764 - val_accuracy: 0.9238
Epoch 8/10
422/422 [==============================] - 1s 1ms/step - loss: 0.3350 - accuracy: 0.9046 - val_loss: 0.2721 - val_accuracy: 0.9237
Epoch 9/10
422/422 [==============================] - 1s 1ms/step - loss: 0.3296 - accuracy: 0.9062 - val_loss: 0.2673 - val_accuracy: 0.9253
Epoch 10/10
422/422 [==============================] - 0s 1ms/step - loss: 0.3247 - accuracy: 0.9073 - val_loss: 0.2637 - val_accuracy: 0.9265
```

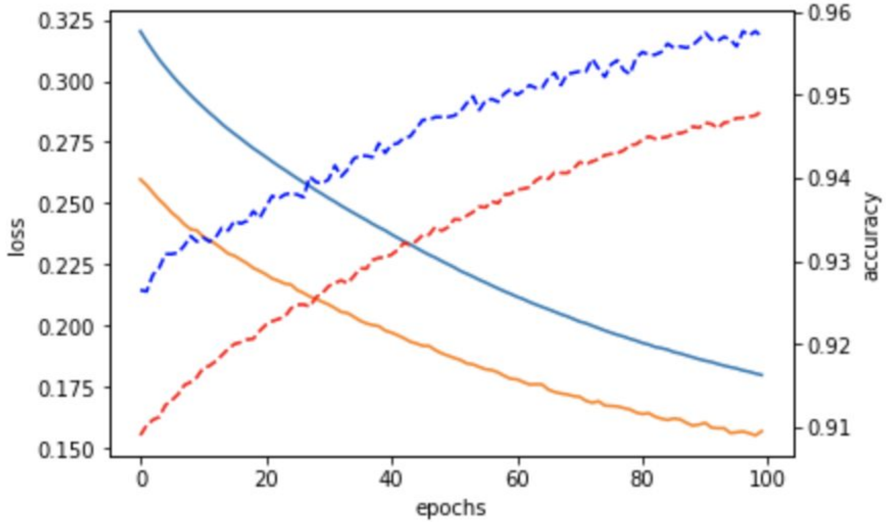
Deep Learning Libraries - Keras example

```
score = model.evaluate(X_test, y_test, verbose=0)
print("Test loss: {:.3f}".format(score[0]))
print("Test Accuracy: {:.3f}".format(score[1]))
```

```
Test loss: 70.345
Test Accuracy: 0.782
```

Deep Learning Libraries - Keras example

```
history_callback = model.fit(X_dev, y_dev, batch_size=128,
                             epochs=100, verbose=1,
                             validation_split=.1)
hist = pd.DataFrame(history_callback.history)
fig,ax = plt.subplots()
ax.plot(hist.index, hist["loss"])
ax.plot(hist.index, hist["val_loss"])
ax.set_ylabel("loss")
ax2 = ax.twinx()
ax2.plot(hist.index, hist["accuracy"], 'r--')
ax2.plot(hist.index, hist["val_accuracy"], 'b--')
ax2.set_ylabel("accuracy")
ax.set_xlabel("epochs")
```



Deep Learning Libraries - Keras example

```
def make_model(optimizer="adam", hidden_size=32):
    model = Sequential([
        Dense(hidden_size, input_shape=(784,)),
        Activation('relu'),
        Dense(10),
        Activation('softmax'),
    ])
    model.compile(optimizer=optimizer,loss="categorical_crossentropy",
                  metrics=['accuracy'])
    return model
clf = KerasClassifier(make_model)
param_grid = {'epochs': [1, 5, 10],  # epochs is fit parameter, not in make_model!
              'hidden_size': [32, 64, 256]}
grid = GridSearchCV(clf, param_grid=param_grid, return_train_score=True)
grid.fit(X_dev, y_dev)
```

https://amueller.github.io/COMS4995-s20/slides/aml-19-convolutional-nets/#12

Deep Learning Libraries - Keras example

```
res = pd.DataFrame(grid.cv_results_)
res.pivot_table(index=["param_epochs", "param_hidden_size"],
                values=['mean_train_score', "mean_test_score"])
```

param_epochs	param_hidden_size	mean_test_score	mean_train_score
1	32	0.741867	0.741058
	64	0.792950	0.794504
	256	0.863850	0.864583
5	32	0.883517	0.884692
	64	0.899267	0.901829
	256	0.912267	0.916233
10	32	0.903417	0.906396
	64	0.912550	0.916312
	256	0.926700	0.931850

Overfitting in Deep Learning Models

Overfitting in Deep Learning Models

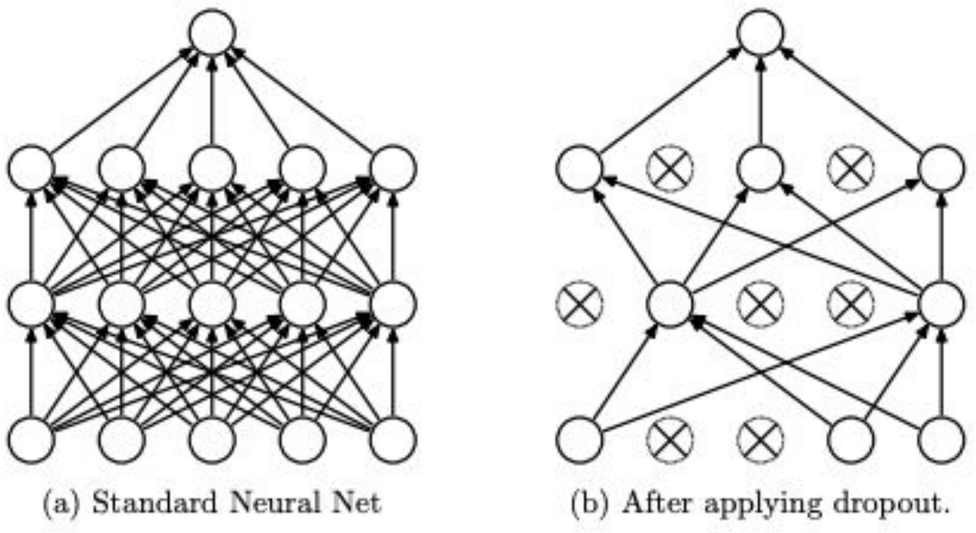
- Dropout
- Batch Normalization

Dropout

Overfitting in Deep Learning Models - Dropout

- Generally ensemble techniques have been good at preventing overfitting in ML methods
- However, training multiple DL models with different architectures and using them for inference is very expensive and time consuming
- Dropout offers a smart way to emulate this behavior during the neural network training process
- Generally, a dropout strategy with ReLU activation function works well in practice

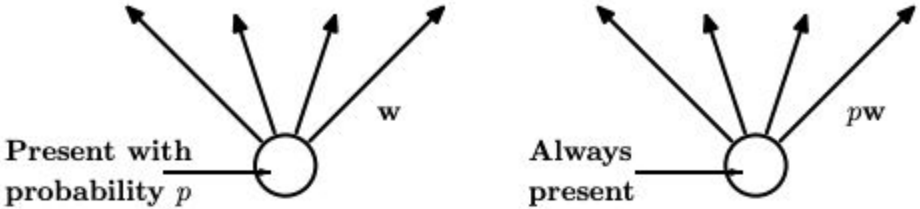
Overfitting in Deep Learning Models - Dropout



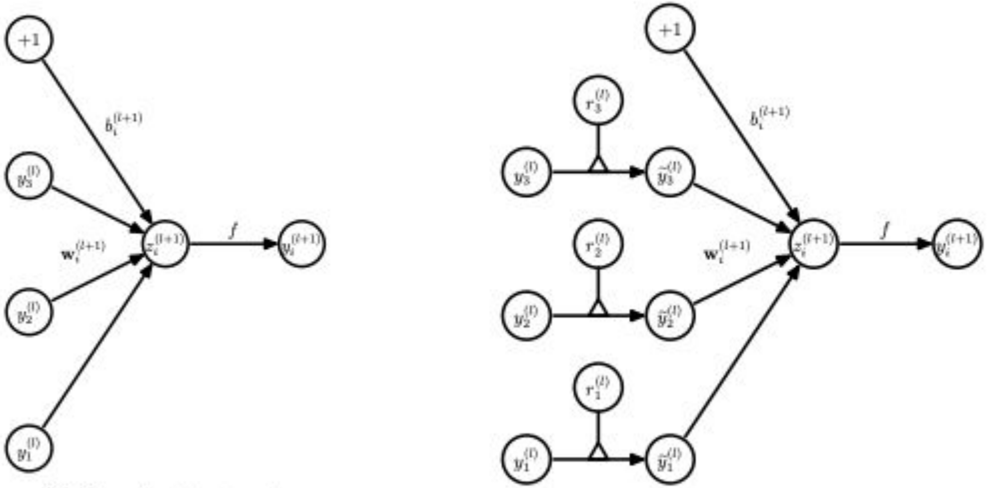
https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

Overfitting in Deep Learning Models - Dropout

Prediction



(a) At training time (b) At test time



(a) Standard network (b) Dropout network

Training

https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

Overfitting in Deep Learning Models - Dropout Example

```
model_dropout = Sequential([
    Dense(1024, input_shape=(784,), activation='relu'),
    Dropout(.5),
    Dense(1024, activation='relu'),
    Dropout(.5),
    Dense(10, activation='softmax'),
])
model_dropout.compile("adam", "categorical_crossentropy", metrics=['accuracy'])
history_dropout = model_dropout.fit(X_dev, y_dev, batch_size=128,
                            epochs=20, verbose=1, validation_split=.1)
```

https://amueller.github.io/COMS4995-s20/slides/aml-20-advanced-nets/#20

Overfitting in Deep Learning Models - Dropout Example

```
score = model_dropout.evaluate(X_test, y_test, verbose=0)
print("Test loss: {:.3f}".format(score[0]))
print("Test Accuracy: {:.3f}".format(score[1]))
```

```
Test loss: 24.049
Test Accuracy: 0.953
```

Batch Normalization

Overfitting in DL Models - Batch Normalization

- The distribution of each layer’s inputs changes during training, as the parameters of the previous layers change.
- This leads to longer training times and choosing initial weights carefully.
- This is referred as internal covariate shift, and is generally addressed by **batch normalization**
- Batch normalization involves normalizing the inputs to the layers for each training mini-batch and is applied per dimension of the input vector.

Overfitting in DL Models - Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

https://arxiv.org/pdf/1502.03167.pdf

Overfitting in DL Models - Batch Normalization Example

```
model_bn = Sequential([
    Dense(1024, input_shape=(784,)),
    BatchNormalization(),
    Activation('relu'),
    Dropout(.5),
    Dense(1024),
    BatchNormalization(),
    Activation('relu'),
    Dropout(.5),
    Dense(10, activation='softmax'),
])
model_bn.compile("adam", "categorical_crossentropy", metrics=['accuracy'])
history_dropout = model_bn.fit(X_dev, y_dev, batch_size=128,
                               epochs=20, verbose=1, validation_split=.1)
```

Overfitting in DL Models - Batch Normalization Example

```
score = model_bn.evaluate(X_test, y_test, verbose=0)
print("Test loss: {:.3f}".format(score[0]))
print("Test Accuracy: {:.3f}".format(score[1]))
```

```
Test loss: 232.494
Test Accuracy: 0.872
```

Questions?