Data 188: Introduction to Deep Learning

Computer Vision: Part 2

Speaker: Eric Kim
Lecture 13 (Week 07)
2026-03-03, Spring 2026. UC Berkeley.

Acknowledgement: slides are based on CMU's Deep Learning Systems course (10-414/714, Fall 2025). Thanks to: Prof. Zico Kolter, Prof. Tianqi Chen, Prof. Tim Dettmers.

Announcements

- HW2 is out!
- Midterm in 1 week!
  - "Midterm Study Guide" released (see Edstem post)

Outline

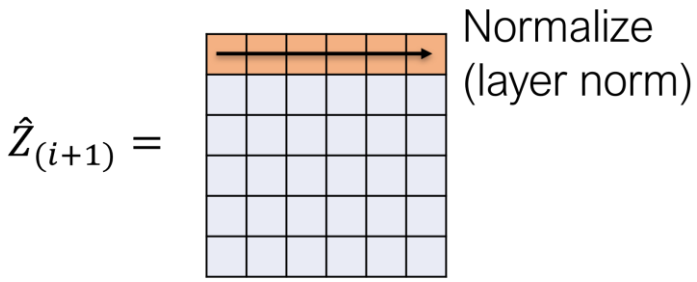
Activation normalization for Convnets

Object detection case study: YOLO ("You Only Look Once")

Activation Normalization for Convnets

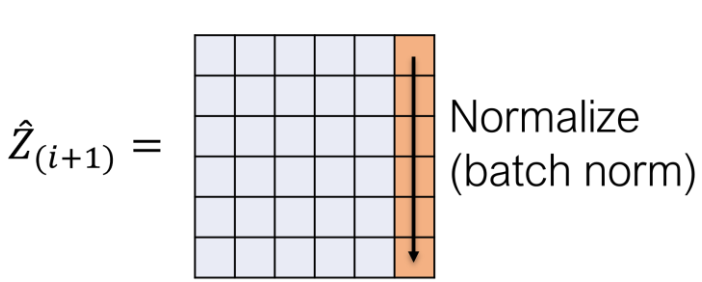
Recall: for MLP networks, we used activation normalization layers (LayerNorm, BatchNorm). Ex: Linear -> **LayerNorm** -> Relu.

For Convnets, our activations are now spatial. How to perform LayerNorm/BatchNorm on spatial feature maps?



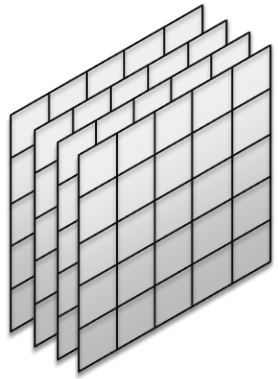
Shape: [batchsize, dim_feats]

LayerNorm1d



Shape: [batchsize, dim_feats]

BatchNorm1d



Shape: [batchsize, channels, h, w]

?

(optional*) LayerNorm2d

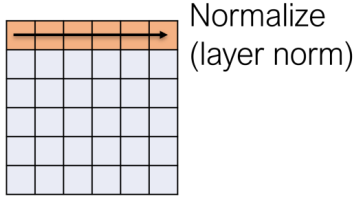
*not in scope for this class

LayerNorm2d calculates per-sample mean/var stats. Reduction is done across [C, h, w].

Affine shift params gamma, beta have shape=[C, h, w].

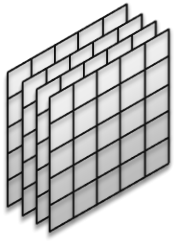
$$Z = \frac{X - \mathbf{E}[X]}{\sqrt{\mathbf{Var}[X] + \epsilon}} \circ \gamma + \beta$$

```
def ln2d_fwd(X: Tensor, gamma: Tensor, beta: Tensor, eps: float) -> Tensor:
    # X.shape: [B, C, h, w]
    # Calculate per-sample mean/var
    # mean,var shape=[B]
    mean = torch.mean(X, dim=[1, 2, 3])
    var = torch.var(X, dim=[1, 2, 3], unbiased=False)
    # normalize input via mean/var
    out = (
        (X - mean[:, None, None, None])
        / (torch.sqrt(var[:, None, None, None] + eps))
    )
    # affine shift. weight,bias shape=[C, h, w]
    out = out * gamma[None, :, :, :] + beta[None, :, :, :]
    return out
```

$\hat{Z}_{(i+1)} =$ 

Shape: [batchsize, dim_feats]

LayerNorm1d



Shape: [batchsize, channels, h, w]

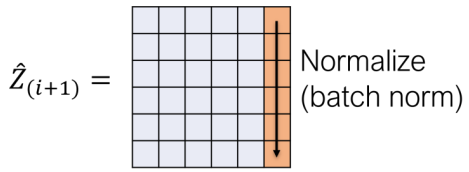
Note: some disagreement on how to do layernorm on spatial feature maps. Calculate mean over [C, h, w], or just over [C]? Ex: ConvNeXt's LayerNorm "channels_last" vs "channels_first". Interesting threads: "LayerNorm, what is going on?", "Improve documentation for LayerNorm..."

BatchNorm2d

BatchNorm2d calculates per-channel mean/var stats. Reduction is done across [B, h, w].

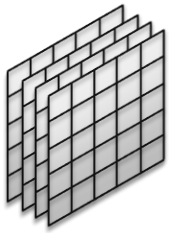
$$Z = \frac{X - \mathbf{E}[X]}{\sqrt{\mathbf{Var}[X] + \epsilon}} \circ \gamma + \beta$$

Affine shift params gamma, beta have shape=[channels].



Shape: [batchsize, dim_feats]

BatchNorm1d



Shape: [batchsize, channels, h, w]

```
def bn2d_fwd_train(X: Tensor, gamma: Tensor, beta: Tensor, eps: float) -> Tensor:
    # X.shape: [B, C, h, w]
    # Calculate per-channel mean/var
    # mean,var shape=[C]
    mean = torch.mean(X, dim=[0, 2, 3])
    var = torch.var(X, dim=[0, 2, 3], unbiased=False)
    # normalize input via mean/var
    out = (
        (X - mean[None, :, None, None])
        / (torch.sqrt(var[None, :, None, None] + eps))
    )
    # affine shift. weight,bias shape=[C]
    out = out * gamma[None, :, None, None] + beta[None, :, None, None]
    # (update running_mean/var not shown)
    return out
```

`bn2d_fwd_test()` uses running_mean/var to normalize X, not cur-batch mean/var, exactly like BatchNorm1d

* Some disagreement. In any case, LayerNorm2d is rarely used for convnets, BatchNorm2d is much more common, so maybe a moot point.

Activation Normalization: 1d vs 2d

	LayerNorm1d	BatchNorm1d	LayerNorm2d*	BatchNorm2d
Input shape	[B, d]	[B, d]	[B, C, H, W]	[B, C, H, W]
Normalization Group (ie mean/var reduction axes)	[d]	[B]	[C, H, W]*	[B, H, W]
Dependent on batch size?	No	Yes: requires batchsize>1	No	Yes: requires batchsize > 1.
Training time same as test time?	Yes	No Train: update running mean/var Test: use running mean/var	Yes	No Train: update running mean/var Test: use running mean/var
Mean/Var shape	[B]	[d]	[B]*	[C]
Affine (gamma, beta) params shape	[d]	[d]	[C, H, W]*	[C]

Spatial Activation Normalization: viz

There are other ways to do spatial activation normalization! Try whichever works best for your task.

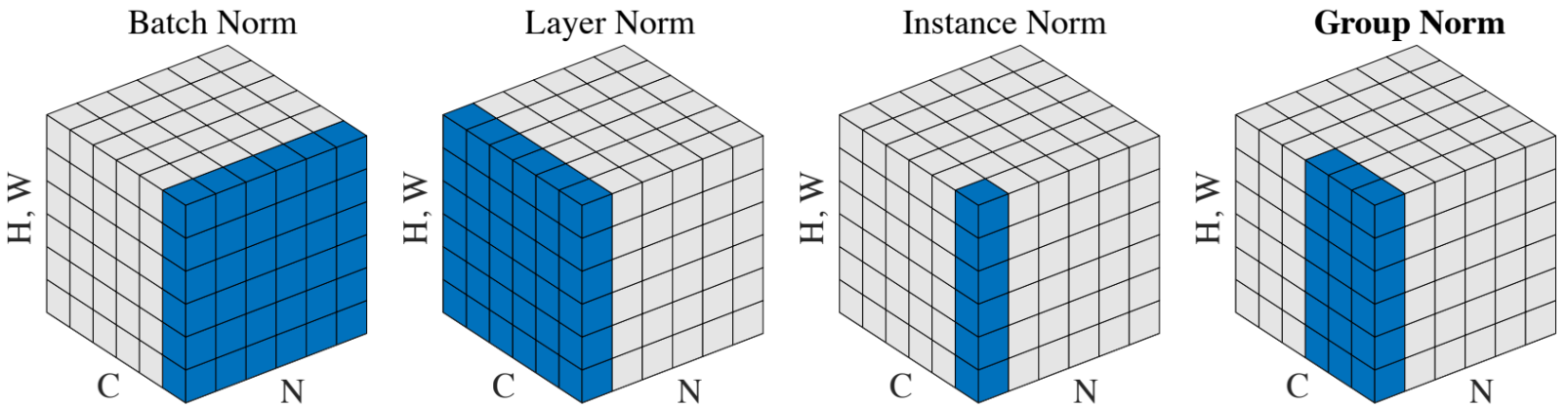


Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

As of 2026, it seems that BatchNorm2d is still popular for convnets. However, LayerNorm is popular for transformer models + NLP tasks.

(optional) interesting discussion on BatchNorm: "When should BatchNorm be used and when should LayerNorm be used?"

https://arxiv.org/abs/1803.08494

YOLO: "You Only Look Once" (2015)

A Convnet-based object detection model.

At the time, a very fast (low-latency) detector with good quality. Simple approach too!

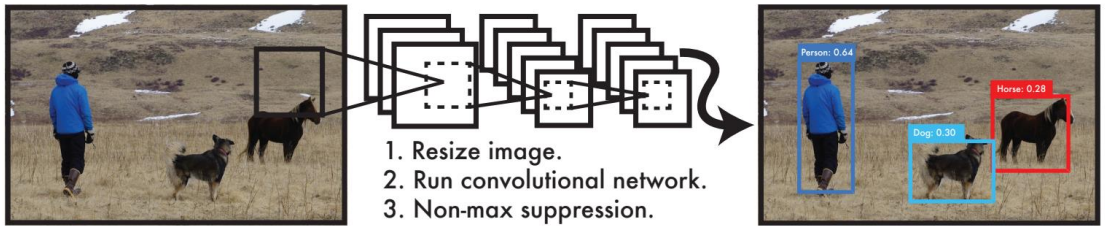


Figure 1: The YOLO Detection System. Processing images with YOLO is simple and straightforward. Our system (1) resizes the input image to 448×448, (2) runs a single convolutional network on the image, and (3) thresholds the resulting detections by the model's confidence.

https://arxiv.org/abs/1506.02640

YOLO: Examples

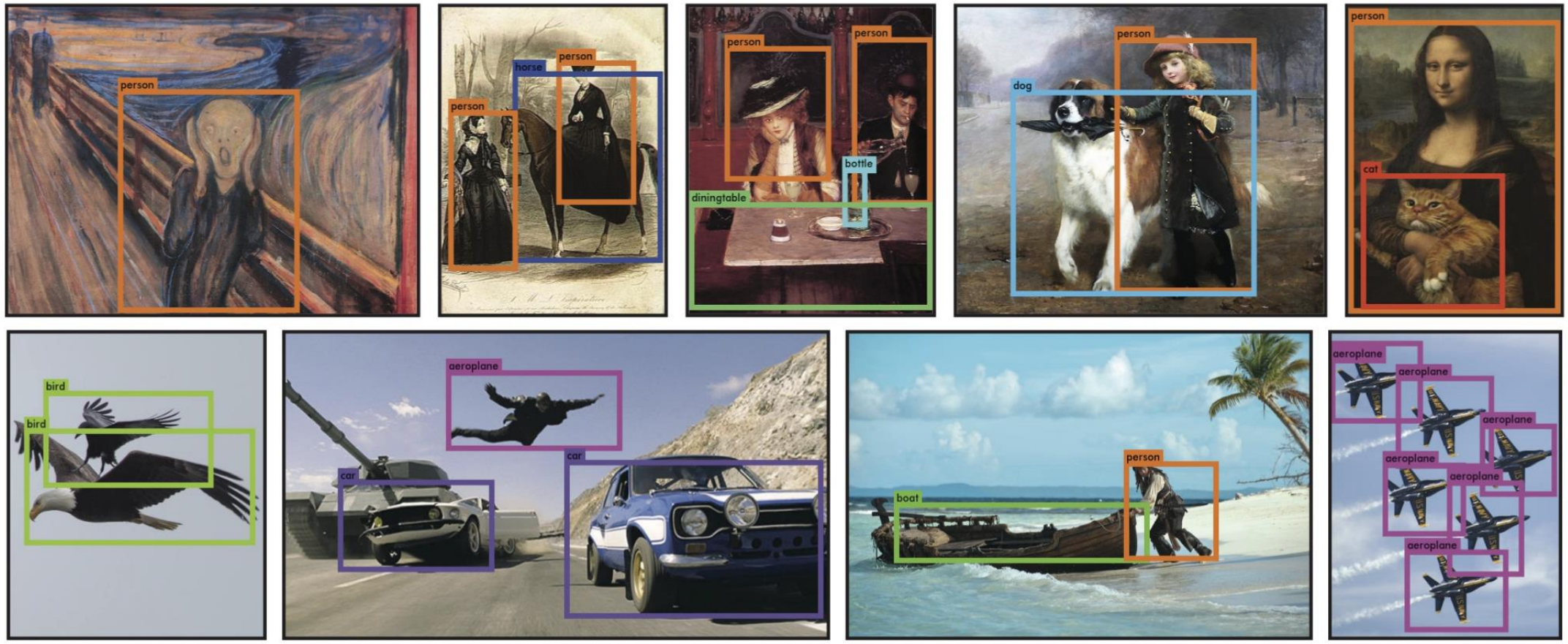


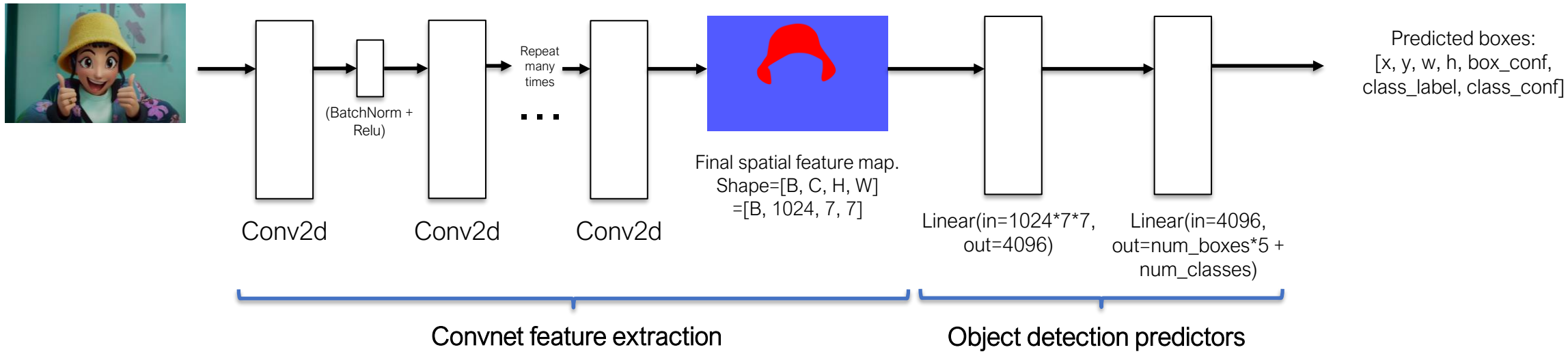
Figure 6: Qualitative Results. YOLO running on sample artwork and natural images from the internet. It is mostly accurate although it does think one person is an airplane.

YOLO approach: high level

(Step 1) Feature extraction. Pass input image [C=3, H=448, W=448] through a convnet.

(Step 2) Detection predictions. From final spatial feature map, predict the bounding boxes [x, y, w, h, box_confidence] (and class labels) via a Linear layer.

(Step 3) Postprocessing. Apply methods like non-maximum suppression (NMS) to remove overlapping predictions. (not visualized below)



num_boxes is a hyperparameter that controls how many boxes to predict in each image. (more details later)

YOLO: image grid

Divide image into S x S grid (S=7).

Note: B is not batchsize!

Each grid cell predicts B boxes (B=2), where each box prediction is: [x, y, w, h, box_conf]

Total number of predicted boxes: S*S*B (7*7*2=98)

Each grid cell also predicts class logits. Important: each grid cell only performs one classification prediction, even though each grid cell predicts multiple boxes! One weakness in approach.

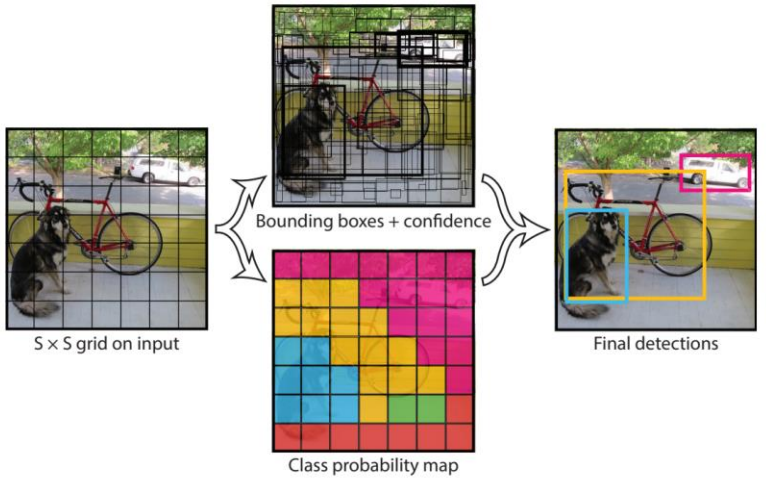
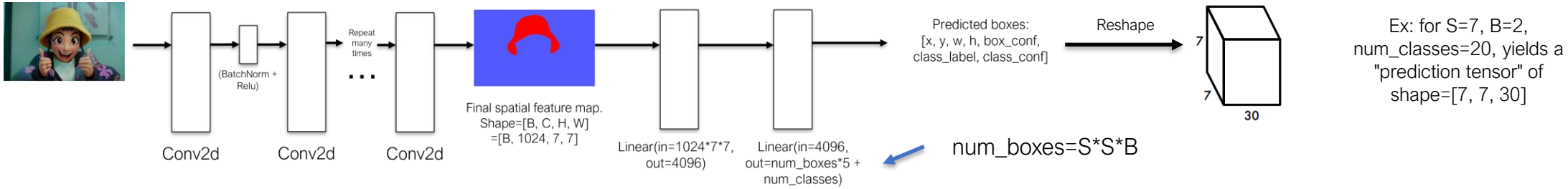
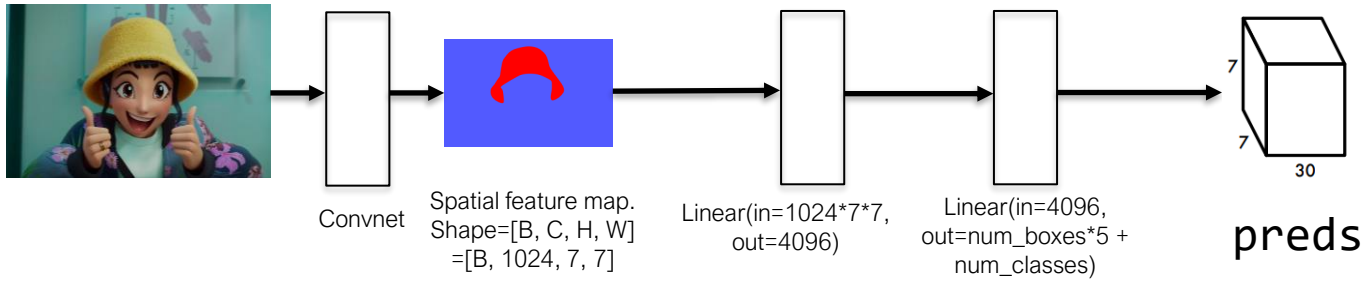


Figure 2: The Model. Our system models detection as a regression problem. It divides the image into an $S \times S$ grid and for each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities. These predictions are encoded as an $S \times S \times (B * 5 + C)$ tensor.



YOLO: prediction tensor



We design this `preds` tensor to have a specific interpretation

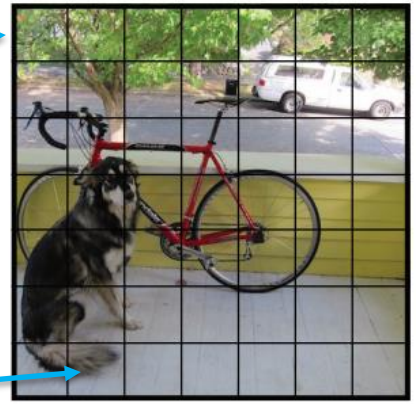
```
# Pred boxes at cell (0, 0)
Preds[0,0,:] = [x1,y1,w1,h1,box_conf1, x2,y2,w2,h2,box_conf2, logit_class1, logit_class2, ..., logit_class20]
```

Box1 preds

Box2 preds

Classifier preds

Recall: B=2 (two predicted boxes per grid cell)

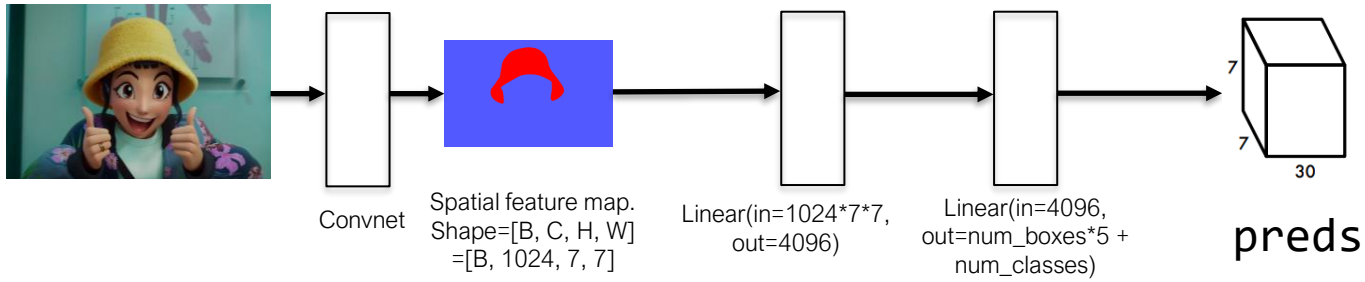


S × S grid on input

```
# Pred boxes at cell (6, 1)
Preds[6,1,:] = [x1,y1,w1,h1,box_conf1, x2,y2,w2,h2,box_conf2, logit_class1, logit_class2, ..., logit_class20]
```

YOLO: classification loss

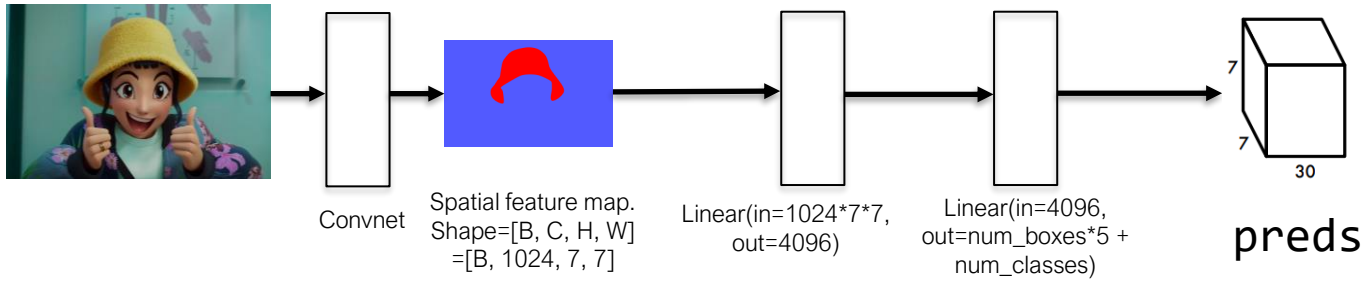
During training, we can add a standard classification loss (eg CrossEntropyLoss) to the relevant parts of the prediction tensor:




```
# Recall: for B=2, num_classes=20, preds for cell (i, j) is arranged as:
preds[i,j,:] = [x1,y1,w1,h1,box_conf1, x2,y2,w2,h2,box_conf2, logit_class1, logit_class2, ..., logit_class20]

cls_loss_layer = CrossEntropyLoss(num_classes=20)

# index+reshape cls preds for loss function
pred_cls = Reshape(preds[:, :, B*5:], [S*S, num_classes])

# gt_cls has shape=[S*S, num_classes]
cls_loss = cls_loss_layer(preds_cls, gt_cls)
```

YOLO: box regression loss

During training, we add a standard regression loss (eg MSELoss) to the relevant parts of the prediction tensor:

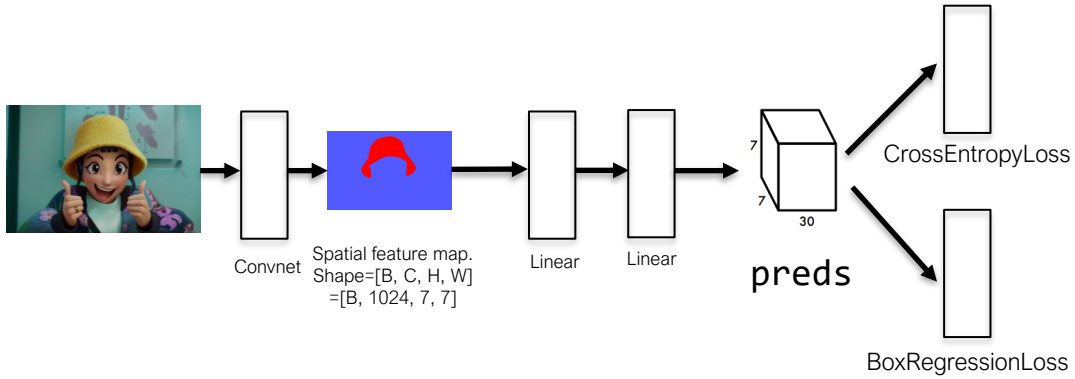



```
# Recall: for B=2, num_classes=20, preds for cell (i, j) is arranged as:
preds[i,j,:] = [x1,y1,w1,h1,box_conf1, x2,y2,w2,h2,box_conf2, logit_class1, logit_class2, ..., logit_class20]

box_loss_layer = MSELoss()

# index box preds for loss function
preds_box = preds[:, :, :B*5]

# gt_box has shape=[S, S, 5]
box_loss = box_loss_layer(preds_box, gt_box)
```

YOLO: cls + box loss

The total loss is a combination of the classification and box regression loss

```
# Recall: for B=2, num_classes=20, preds for cell (i, j) is arranged as:
preds[i,j,:] = [x1,y1,w1,h1,box_conf1, x2,y2,w2,h2,box_conf2, logit_class1, logit_class2, ..., logit_class20]

cls_loss_layer = CrossEntropyLoss(num_classes=20)
# index+reshape cls preds for loss function
pred_cls = Reshape(preds[:, :, B*5:], [S*S, num_classes])
# gt_cls has shape=[S*S, num_classes]
cls_loss = cls_loss_layer(preds_cls, gt_cls)

box_loss_layer = MSELoss()
# index box preds for loss function
preds_box = preds[:, :, :B*5]
# gt_box has shape=[S, S, 5]
box_loss = box_loss_layer(preds_box, gt_box)

# w_cls, w_box are scalar weights (hyperparams).
total_loss = w_cls * cls_loss + w_box * box_loss
total_loss.backward()  # the magic of backprop + deep learning!
```

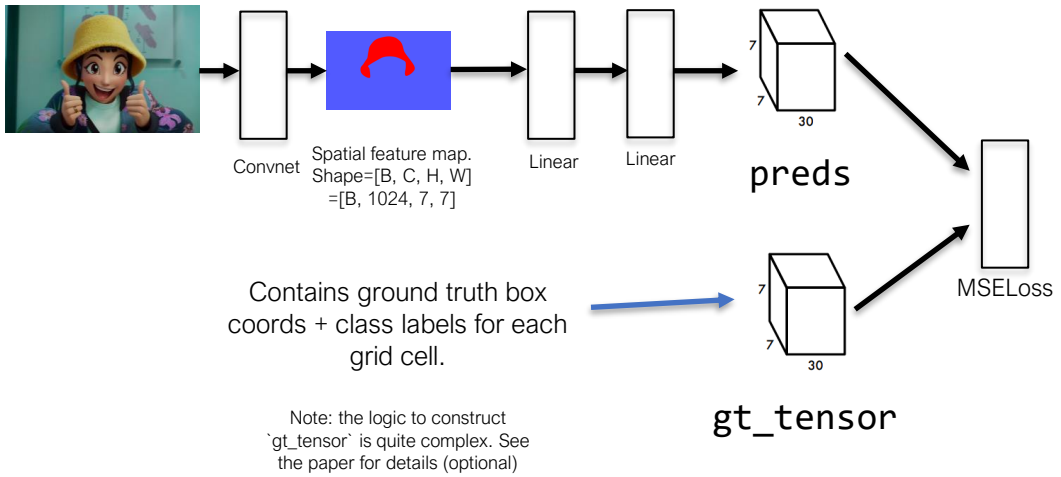


YOLO: cls + box loss (streamlined)

In the paper, they streamlined the total loss by calculating MSE between the `preds` tensor and the `gt_tensor`:

```
# preds.shape=[batchsize, S, S, (B*5)+num_classes)
# Recall: for B=2, num_classes=20, preds for cell (i, j) is arranged as:
preds[i,j,:] = [x1,y1,w1,h1,box_conf1, x2,y2,w2,h2,box_conf2, prob_class1, prob_class2, ..., prob_class20]

# shape=[batchsize, S, S, (B*5)+num_classes]
gt_tensor = construct_gt_tensor(gt_labels_and_boxes)

loss_layer = MSELoss()
total_loss = MSELoss(preds, gt_tensor)
total_loss.backward()
```

One interesting takeaway: shows that doing classification via MSE regression can work! (but, probably better to do CrossEntropyLoss)



YOLO: gt_cls, gt_box details

Constructing `gt_cls, gt_box` is crucial ("the devil is in the details"), and contains most of the complexity.

Example: "assign" each ground-truth box to a grid cell(s). If a GT box's center lands within a grid cell (i, j), then that grid cell is "responsible" for predicting that GT box.

=> `gt_box[i, j, :]` contains the GT box

```
# Recall: for B=2, num_classes=20, preds for cell (i, j) is arranged as:
preds[i,j,:] = [x1,y1,w1,h1,box_conf1, x2,y2,w2,h2,box_conf2, logit_class1, logit_class2, ..., logit_class20]

cls_loss_layer = CrossEntropyLoss(num_classes=20)
# index+reshape cls preds for loss function
pred_cls = Reshape(preds[:, :, B*5:], [S*S, num_classes])
# gt_cls has shape=[S*S, num_classes]
cls_loss = cls_loss_layer(preds_cls, gt_cls)

box_loss_layer = MSELoss()
# index box preds for loss function
preds_box = preds[:, :, :B*5]
# gt_box has shape=[S, S, 5]
box_loss = box_loss_layer(preds_box, gt_box)
```

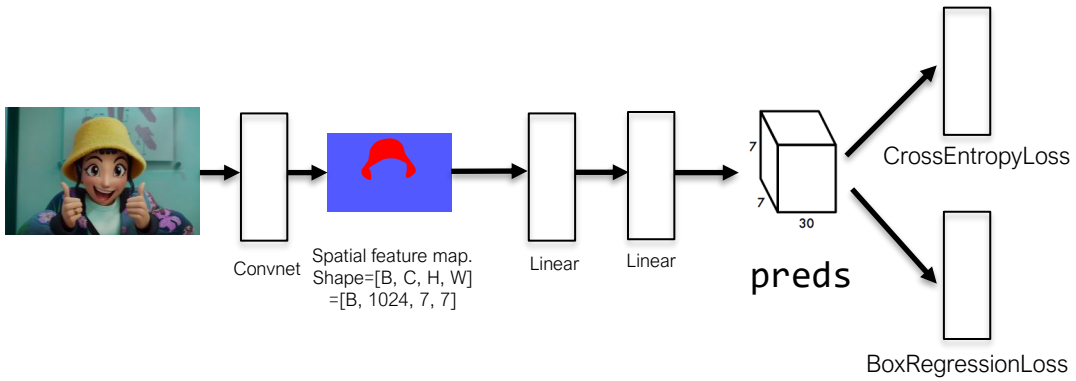
For now, we'll skip the nitty-gritty details. Feel free to read the paper for the gory details!

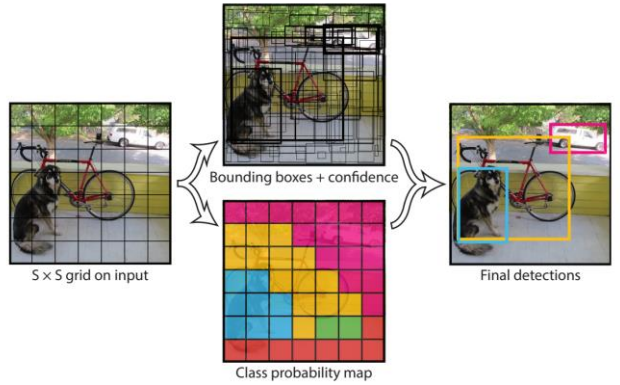
YOLO: takeaways

Convnet flexibility: we can extend Convents to perform tasks other than classification (ex: box regression).

Modularity: A convnet is abstracted as a "spatial feature map extractor", and is one component of the object detection model.

End-to-end training: since each step of YOLO is differentiable, we can train the model end-to-end. Notably, we can jointly train the convnet ("feature extractor") and the detection heads (classification, box regression)!



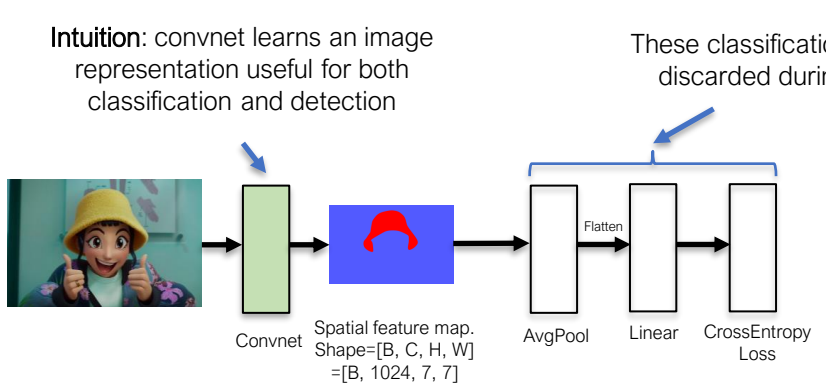


YOLO: transfer learning

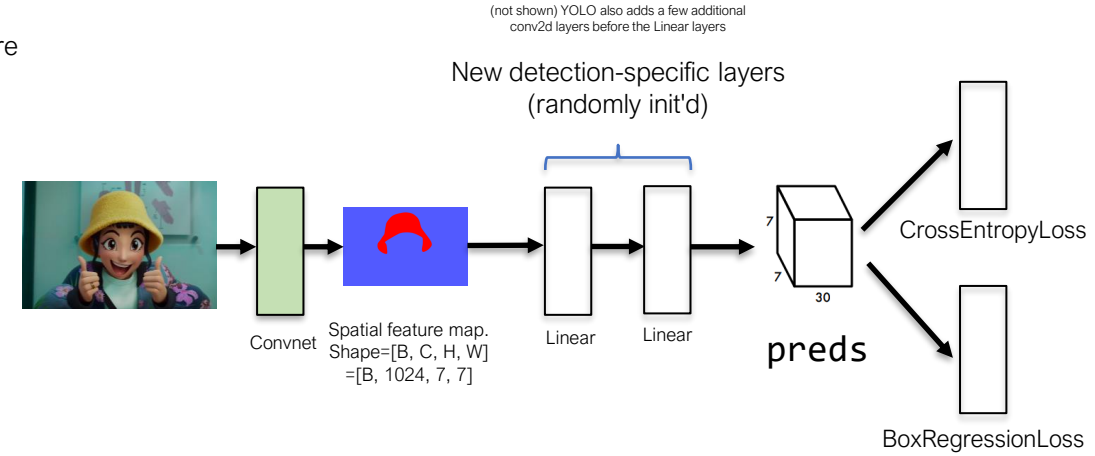
Rather than start with a randomly-initialized convnet, YOLO does a "two stage" training process.

Pretrain: train convnet on an image classification dataset (ex: ImageNet-1k).

Finetune: attach detection-specific layers (randomly init'd). Train entire model on object detection dataset.



Pretrain Convnet on image classification dataset.

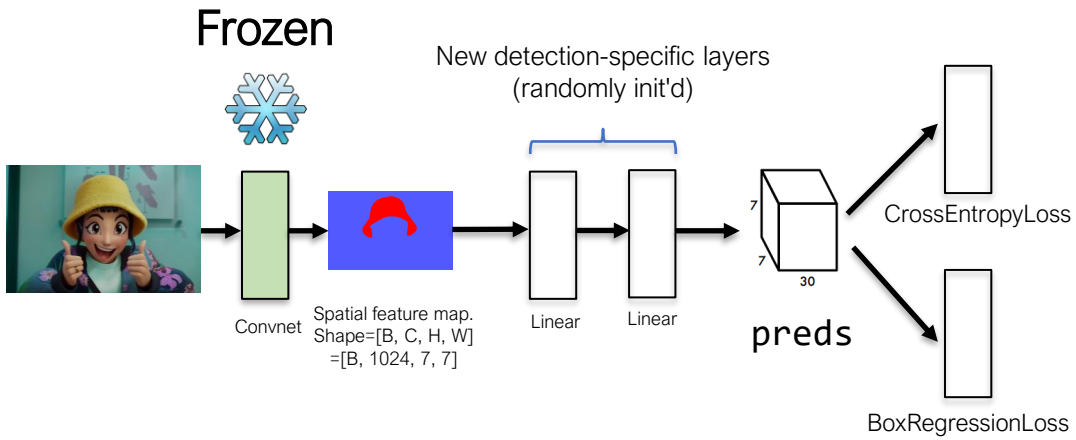


Attach new task heads/layers, and **finetune** on object detection dataset

Layer freezing

During second-stage finetuning, if compute is limited, it's common to keep the convnet layers fixed ("**frozen**"), and only learn the detection-specific layers.

Reduces required computation and (GPU) memory (no need to compute or store adjoints for frozen layers).



Attach new task heads/layers, and **finetune** on object detection dataset

A few related tricks:

- Reduce learning rate for conv layers (say, 10x lower)
- Freeze only first few layers of convnet

Transfer Learning

Transfer learning (aka "pretrain + finetune") is an effective, widely-used technique in deep learning.

NLP example: pretrain a large language model on Wikipedia text articles ("next token prediction task"), then finetune on text sentiment analysis classification task.

Computer vision: pretrain a convnet on ImageNet-1k, then finetune for image segmentation.

Intuition: features that are useful for one task (ex: image classification) are often also useful for other tasks (ex: object detection, segmentation, or classification on other class taxonomies).